NAVAL POSTGRADUATE SCHOOL
Monterey , California

THESIS

A PERFORMANCE ANALYSIS OF VIEW MATERIALIZATION STRATEGIES FOR GENERAL EXPRESSIONS

by
Curtis G. Barefield Jr.

September 1991

Thesis Advisor Magdi N. Kamel

Approved for public release; distribution is unlimited.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b OFFICE SYMBOL *(If applicable)* 55	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c ADDRESS *(City, State, and ZIP Code)* Monterey, CA 93943-5000	7b ADDRESS *(City, State, and ZIP Code)* Monterey, CA 93943-5000
8a NAME OF FUNDING/SPONSORING ORGANIZATION 8b OFFICE SYMBOL *(If applicable)*	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS *(City, State, and ZIP Code)*	10 SOURCE OF FUNDING NUMBERS Program Element No / Project No / Task No / Work Unit Accession Number

11 TITLE *(Include Security Classification)*
A PERFORMANCE ANALSYIS OF VIEW MATERIALIZATION STRATEGIES FOR GENERAL EXPRESSIONS

12 PERSONAL AUTHOR(S) Barefield, Curtis Gus Jr.

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT *(year, month, day)*	15 PAGE COUNT
Master's Thesis	From To	1991 September	110

16 SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 COSATI CODES			18 SUBJECT TERMS *(continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUBGROUP	View processing strategies, semi-materialization, test database, view materialization strategies

19 ABSTRACT *(continue on reverse if necessary and identify by block number)*

Efficient processing of views is critical to many real world applications such as surveillance systems which support military applications. This thesis compares the performance of three view materialization strategies: semi-materialization, full materialization and query modication. This thesis first develops a program that generates databases according to user specification. Second the generated databases are used to conduct an empirical study on the three view materialization strategies using select-project-join and general expression views. The results of the study indicate that for select-project-join view definitions, semi-materialization performed best for higher values of l, fv, and all values of fq with the database stored on hard disk. Full materialization performed best for lower values of P, l, and all values of fv with the database stored in RAM. The results also indicate that the semi-materialization strategy is the best view processing method for general expressions.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☒ SAME AS REPORT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL Magdi N. Kamel	22b TELEPHONE *(Include Area code)* (408) 646-2494 22c OFFICE SYMBOL AS / KA

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Approved for public release; distribution is unlimited.

A Performance Analysis of View Materialization Strategies for General Expressions

by

Curtis G. Barefield Jr.
Lieutenant, United States Navy
B.S., Wayland Baptist University

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN INFORMATION SYSTEMS

from the

ABSTRACT

Efficient processing of views is critical to many real world applications, particularly real time applications such as surveillance systems which support military applications. This thesis compares the performance of three view materialization strategies: semi-materialization, full materialization and query modification. This thesis first develops a program that generates databases according to user specification. Second the generated databases are used to conduct an empirical study on the three view materialization strategies using select-project-join and general expression views. The results of the study indicate that for select-project-join view definitions, semi-materialization performed best for higher values of *P*, lower values of *l*, *fv* and all values of *fq* with the database stored on hard disk. Full materialization performed best for lower values of *P*, *l*, and all values of *fv* with the database stored in RAM. The results also indicate that the semi-materialization strategy is the best view processing method for general expressions.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. BACKGROUND

A database is a computer based record keeping system that contains information used to support an organization's tactical (short range) and strategic (long range) goals. For example, a database for a sales organization could contain customer, employee, sales and inventory data.

Several data models are available to organize the information within the database so that it can be utilized in an efficient manner. One of the most common data models is the relational model. This method organizes data in terms of tables (relations), rows (tuples) and columns (attributes).

Tables can be classified as either base tables or views. A base table is a table that physically exists in its own right. A view maybe thought of as a virtual table, in as much, that it does not (normally) exist within its own right but is instead derived from one or more underlying base tables [Ref. 1]. The view is stored as a definition in the data dictionary and is combined with a user's query to retrieve the requested data from the base tables.

The use of views allows for the structuring and limiting of the information retrieved by a given query. This feature allows the user to receive data that is relevant to the application and limits unauthorized user access to other critical data.

Recently several proposals have considered storing some form of the processed view to eliminate the need to evaluate the view definition from scratch every time it is queried. The first approach, known as full materialization, stores the fully processed view as a physical table. This approach has the advantage of increasing the efficiency of the queries on the view , but incurs an additional expense of maintaining the materialized view. To overcome this problem, a second approach, called semi-materialization, was proposed whereby a partially processed rather than a fully materialized view is stored. This approach redundantly stores data that represents selections and projections of individual relations, thus allowing efficient evaluation of the view definition while being easy to maintain.

View performance processing is directly related to the performance of real time applications such as surveillance systems which support military operations. These systems receive periodic environmental updates from various sensors which are used to evaluate a view. Any delay processing the sensor data, which is typically time sensitive, into usable information could render the information late and unusable.

Faster view processing used in conjunction with real time systems will significantly improve the response time of these systems.

B. OBJECTIVE

The objective of this thesis is to compare empirically the performance of three view processing strategies: query modification, semi-materialization and full materialization. The research attempts to verify the analytical results which have indicated that, in general, the semi-materialization strategy is the best method for processing general expression views [Ref. 2]. To accomplish this goal, this research develops a Data Generation Program to produce test databases according to user specifications. The test databases are then used to compare the performance of three view processing strategies for two view expressions and under different parameter settings, using a simulation program that was developed by Lt Jesse South [Ref. 3]. Performance results were then collected, analyzed and plotted for presentation in this thesis.

C. SCOPE AND METHODOLOGY

This thesis accomplishes the following:

1. Develops a generic database generating program using ANSI C to generate test databases according to user specifications.

2. Compares the performance of three view materialization strategies for select-project-join expressions with the database stored in Random Access Memory (RAM) and hard disk.

3. Tests the three view strategies using general expressions with the database stored on RAM and on hard disk under different parameter settings, collecting the results and comparing them with analytical results.

4. Uses the results to draw conclusions and determine the conditions under which each strategy performs the best.

D. ORGANIZATION OF STUDY

This thesis is organized as follows. Chapter II overviews the three view processing strategies. Chapter III provides a detailed description of the Data Generation Program. Chapter IV presents the performance results of the empirical study and compares them to the results of the analytical study. Chapter V presents conclusions based on the study and suggests areas for future research.

II. VIEW MATERIALIZATION STRATEGIES

The purpose of this chapter is to provide a general overview of the three view materialization strategies - query modification, semi-materialization and full materialization.

A. QUERY MODIFICATION

The conventional method for view processing for queries is query modification. This method stores a view definition in the data dictionary. This view definition is retrieved from the data dictionary when a query is issued on the view and combined with the user query into an equivalent query on the underlying base tables. This query is subsequently processed, and the results returned to its user. Consider the following database schema:

EMP(E#,ENAME,ADDRESS,SALARY,TITLE)

POS(E#,S#,LEVEL)

and the corresponding view definition COMBATSTAFF:

$$\Pi_{e.ENUM,\, e.ENAME,\, e.SALARY}(\sigma_{p.LEVEL>3}(EMP \bowtie POS))$$

Now when a query is issued against COMBATSTAFF:

$$\Pi_{c.ENUM,\, c.ENAME}(\sigma_{c.SALARY>30,000}(COMBATSTAFF))$$

The view mechanism translates the query into the equivalent query on the base relations:

$$\Pi_{e.ENUM,\, e.ENAME}\ (\sigma_{e.salary>30{,}000 \wedge p.LEVEL>3}\,(EMP \bowtie POS))$$

The resulting query is optimized to determine the best access path and then executed.

B. FULL MATERIALIZATION

This method creates an actual table based on the view definition. The resulting table is used to perform user queries, thus avoiding the cost of repeatedly retrieving a view definition and creating equivalent queries on the base relations. This method works quite well for processing queries, but is costly when the frequency of update is high, since the full materialized view must be maintained.

Updates are defined as a transaction which performs a sequence of tuple insertions, tuple deletions, and tuple modifications on a relation(s). Suppose that a set of tuples A is added to a relation and a set of tuples D is deleted from the same relation. The tuple sets A and D represent the net change made to that relation. In that case, a tuple which is inserted and deleted in the same transaction would not appear in either tuple set A or D.

Using this method, the net results of an update transaction could be used as a basis for a differential algorithm to update the materialized view.

In fact this method works quite well using select-project-join expressions because selections and projections can be performed over unions. Using the view definition in the previous section and limiting the updates to the POS relation for simplicity, the view expression becomes:

$$COMBATSTAFF' = COMBATSTAFF - \Pi_{e.ENUM,\, e.ENAME,\, e.SALARY} (p.LEVEL > 3\, (D_1 \bowtie POS)) \cup \Pi_{e.ENUM,\, e.ENAME,\, e.SALARY} (p.LEVEL > 3\, (A_1 \bowtie POS))$$

The above expression shows that the fully materialized view can be maintained by computing the last two expressions and inserting them into or deleting them from the materialized view COMBATSTAFF.

Unfortunately a similar expression can not be derived if a general expression is used in the view definition. At present no efficient differential algorithm exists for performing incremental updates for general expressions. This fact necessitates that a complete re-evaluation of the view expression be accomplished after each update to the base relations. The cost of re-evaluating a fully materialized view can be prohibitive as the frequency of updates for the base relations increase, which is the chief problem associated with this method.

C. SEMI-MATERIALIZATION

This method stores redundant subsets of carefully chosen data from individual base tables. These redundant subsets are stored as actual tables and represent an intermediate state of computing the view. Each subset is a projection and selection of the base table(s) thus making the construction of the view less costly than using the base relations.

The redundant data is clustered on the join attribute(s) which allows for the efficient construction of the view. Updates to the base relations are screened to determine if the update affects the redundant tables. If it does, it is inserted into or deleted from the appropriate redundant tables.

The following redundant subsets would be stored to support this technique:

$$EMP' = \Pi_{e.ENUM,\, e.ENAME,\, e.SALARY}(EMP)$$
$$POS' = \Pi_{p.ENUM}(\sigma_{p.LEVEL>3}(POS))$$

This view is combined with a user query to form an equivalent query on the redundant relations:

$$\Pi_{c.ENUM,\, c.ENAME,\, c.SALARY}(EMP' \bowtie POS')$$

When queried the following equivalent view is created using the redundant tables and the view definition:

$$\Pi_{e'.ENUM,\, e'.ENAME}(\sigma_{e'.SALARY>30{,}000}(EMP' \bowtie POS'))$$

This method becomes more complicated as additional insertions and deletions occur. Since more than one base relation may have been the source of the tuples used in the materialized view, it becomes increasing difficult to determine, when or if a record should be removed from the view.

To alleviate this problem each materialized view must keep a duplicate count of the number of tuples contributed, by each redundant subset, to the tuples in the materialized view when the subsets are joined. The count should be incremented or decremented depending on the transaction until the count becomes zero.

III. DATA GENERATION PROGRAM

The purpose of this chapter is to describe the Data Generation program. According to user specifications the program generates text files that are used subsequently to build the test database. As shown in Figure 1, the program reads control information from a text file created by a user or generated by the simulation program and generates the specified text files. The program allows the user to control the number of records (cardinality of the relation), the data type (ALPHA, NUMERIC or ALPHANUMERIC characters), the size of each field and the number of fields generated for each record.

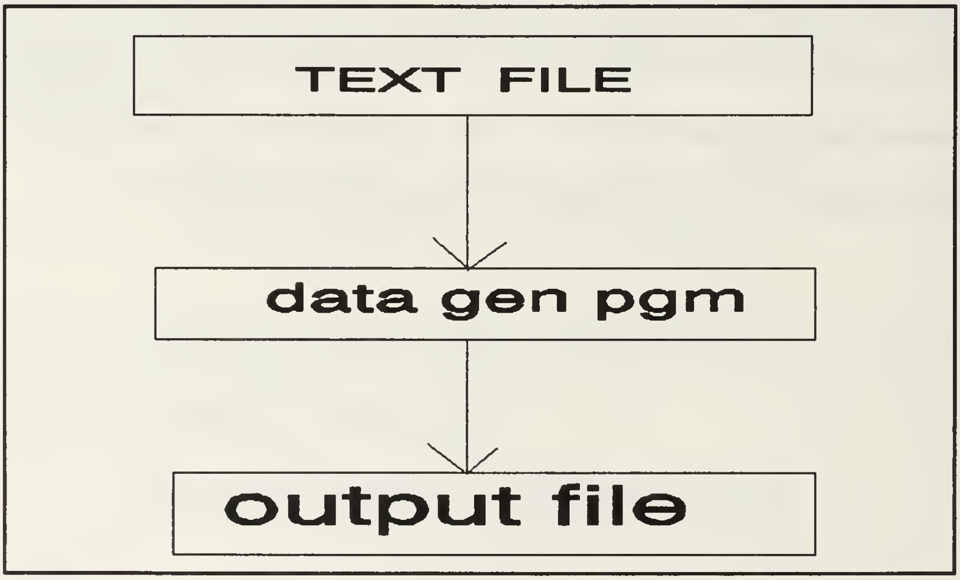

Figure 1: Data generation program data flow overview.

The process to generate the data is hidden from the user by using a fixed format control file as the user interface to the program.

The program is written in ANSI C to increase portability of the source code and to minimize the changes necessary to transfer the program to a mini or mainframe environment. The maximum size of the text file generated by the program is limited only by the secondary storage available on the platform in use.

A. GENERAL DESCRIPTION

The Data Generation program receives control data from the text file "DATA_IN". The information in the control data file is effectively divided into two sections. The first section determines the number of records, fields per record and the name of the output file. The second section defines each field within the record by type of information for the field (Field Type:Alpha, Numeric or Alphanumeric): the number of characters for each attribute (Field Width): the upper and lower bounds for any arrays and the incremental value used for counters. Data Generation program reads the data into a set of linked lists which are passed to the control modules by the main module to create each record.

B. REQUIREMENTS

The requirement for the Data Generation program was based on a user request that a new generic data generating program be written in the C programming language to replace the previous database generating program.

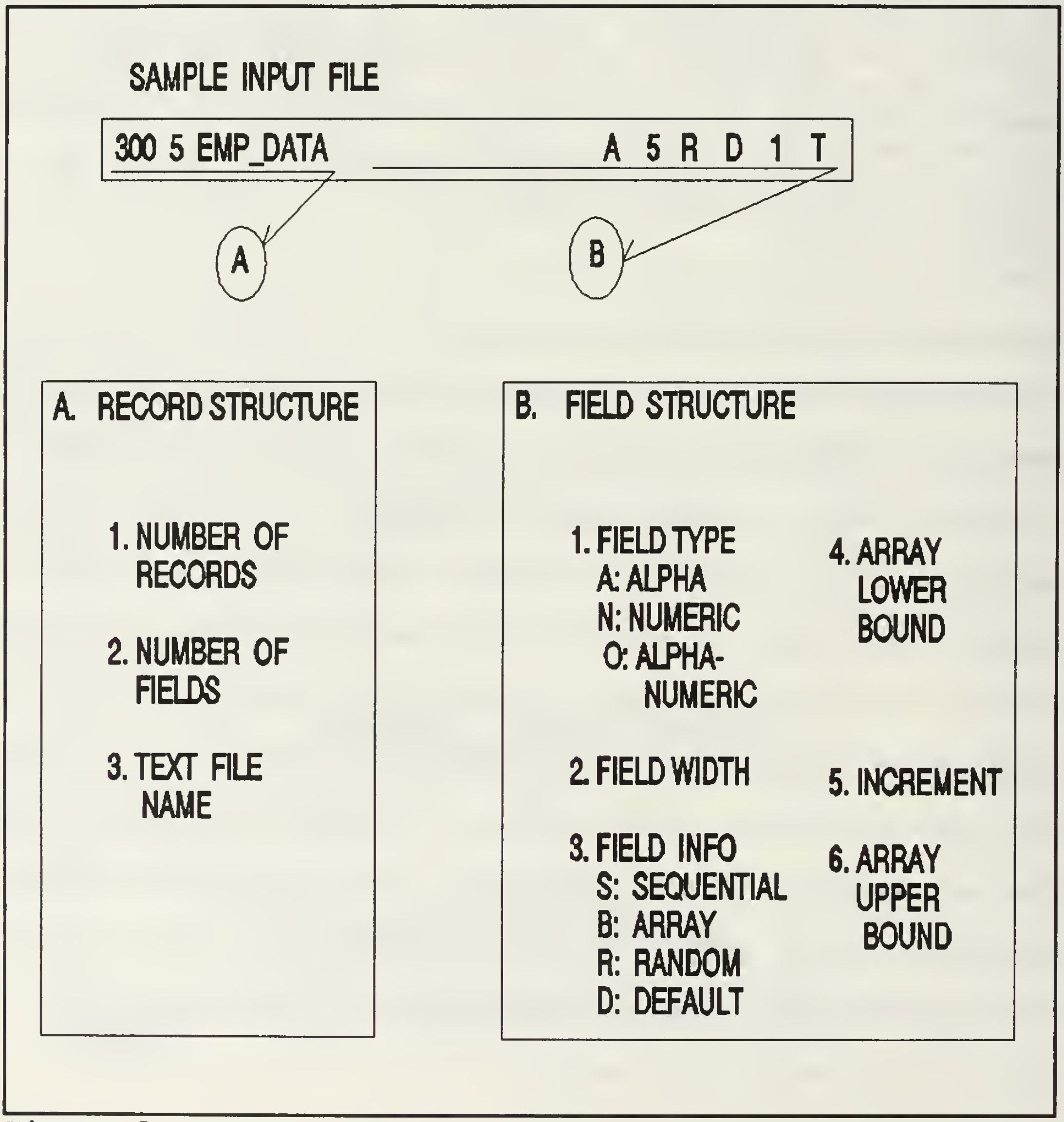

Figure 2: Data generation program parameters and parameter definitions.

The program accepts the following inputs and generates a text file used to create test databases:

1. Number of text files required.
2. Number of records per text file.
3. Name of the text file.
4. Number of fields per record.
5. Size of each field.
6. Type of information in each field.
7. Number of distinct values in each field.
8. Upper and lower limits for the fields.
9. Input reference for randomly generated characters.

To simplify the performance analysis several assumptions were made about the data generated for the test database. The first assumption was that the values for each field in the column were uniformly distributed over the range of values in the column. The second assumption considered each value in a given column to be independent of the values in the other columns.

C. NOTES ON PROGRAM DESIGN

The requirement for maximum program flexibility dictated a "layered" design approach be used, creating individual primitive modules to produce the varied types of output data requested by the user.

To keep the coupling between the modules as loose as possible, the use of global variables is minimized and when feasible, only a single record structure is passed between the called and calling modules.

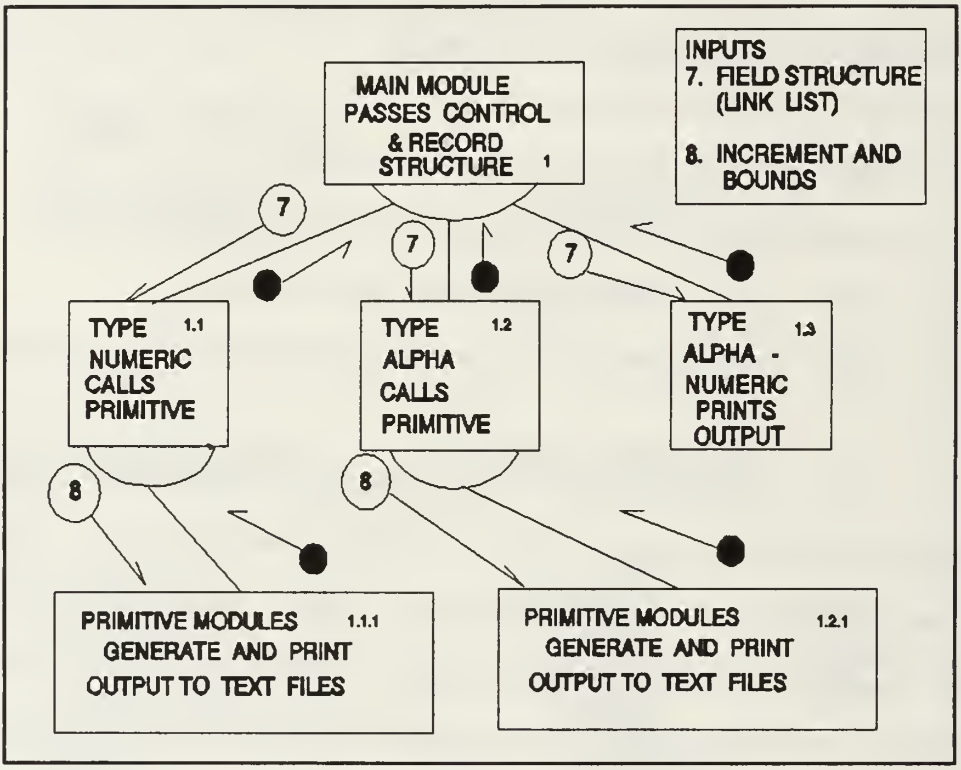

Figure 3: Data generation program control module overview.

Each primitive module prints its output directly to the output text file with the exception of the generate_numeric_array module which returns its numeric output to the random_generator module for conversion to alpha characters, if required.

This method was chosen after trial and error as the best method for facilitating the tracing of data and control flow through the modules.

The rand( ) C library function was used to generate random data. Two C language record structures were used to establish the command language between the data generation program, the control file and the view materialization simulation program.

D. PROGRAM MODULE OVERVIEW

A brief description of each module is provided to clarify the control and data flows that are described in Section E.

1. Main Module

The main module opens and closes the input-output files, loads the control data into the record structures and directs the flow of the control data to the applicable modules for data generation.

2. Time Hack Module

The time hack module uses the system clock to compute the base reference for the generation of random alpha and numeric output values.

3. Type_Numeric Module

The type_numeric module is called by the main module to generate a numeric string, or call the sequential counter, random_generator or the bounded_sequential_array modules.

4. Type_Alpha Module

The type_alpha module is one of three process control modules used to determine the type of characters in a field. The module receives its input in the form of a record structure passed from the main module to generate a string of alpha characters, or to call the random_generator or the bounded_sequential_array modules.

5. Type_Alphanumeric Module

The type_alphanumeric module is the last process control module and generates a single variable length string of alpha and numeric characters when called by the main module.

6. Bounded_Sequential_Array Module

The bounded_sequential_array module, which is called by either the type_alpha or type_numeric modules, receives three numeric values from the calling module. The values determine the array lower bound, the number of array elements and the incremental value of each element. The rand( ) function is used to generate a random index number to select the array element value that is printed in the output file.

7. Random_Generator Module

The random_generator module is called in the same manner as the bounded_sequential_array module. The module determines if a character or numeric value is required, calls the generate_numeric_array module to produce the required value and prints the value or character in the output file.

8. Random_Long_Array Module

The random_long_array module is called by the type_numeric module to produce a random numeric output employing the same rand() function that was used in the bounded_sequential_array module. The module computes the array size and determines if the number of array elements exceeds a preset limit.

The module will compute the output value using the upper bound value and the rand() function to conserve main memory rather than allocating space for the array if the preset limit is exceeded. This method was used to prevent the program from using memory unnecessarily.

9. Counter Module

The counter module is called by the type_numeric module and uses global values to generate up to three independent sequential counters.

10. Generate_Numeric_Array Module

The generate_numeric_array module is called by the random_generator module to produce a second independent bounded array similar to the bounded_sequential_array module except the random numeric output from the module is returned to the calling module for possible conversion to an alpha character.

11. Print Modules

The print modules are all used to send debugging data to a text file called "output.txt" that is controlled by a toggle called "TROUBLESHOOTING".

E. DETAILED DATA AND CONTROL FLOW

The Data Generation Program is called by a batch file which reads the control file "DATA_IN". The input data is formatted to conform to the two record structures declared in the definition section of the program.

Once the input data is loaded into the program, the control file is closed and the output file is opened. The output file name is part of the control file data. Each record structure is read and control is routed to the appropriate control module based on field type (ALPHA "A", NUMERIC "N" and ALPHANUMERIC "O").

The type_numeric module will be used to trace the first data flow through the modules, the second data flow be traced using the type_alpha module and the last data flow will use type_alphanumeric module.

"N" is the field type read by the main module in the attribute record structure. Control and the attribute record structure is passed to the type_numeric module by the main module. The attribute record structure is read by the module to determine the field information (BOUNDED SEQUENTIAL ARRAY "B", RANDOM GENERATOR "R", COUNTER "S", RANDOM LONG ARRAY "X" or DEFAULT "D").

The field information type read by the module is "B" and the bounded_sequential_array module is called. The type_numeric module converts the lower and upper bound character strings to numeric values which are passed to the bounded_sequential_array module along with the incremental data. The module uses the input data to determine array size, the lower bound and increment.

Memory is allocated and the array is filled. The rand() function and the array size are used to compute an random index number to select the array element value to be printed in the applicable field in the output field.

Control is returned to the main module and the next attribute record structure is read. "A" is the next field type read by the main module: control and the attribute record structure is passed to the type_alpha module.

The type_alpha module reads the attribute record structure. "R" is the field information read by the module. The random_generator module is called, the lower bound character and upper bound character strings are read by the type_alpha module. The character strings are converted to numeric values and passed along with the incremental data to the random_generator module.

The random_generator module determines if the integer values represent alpha characters or numeric values. In this case, the values represent the upper case letters "A"(lower bound), "R" (upper bound), and the increment value of 1. The random_generator computes the array size, and passes the values to the generate_numeric_array module to generate the array.

The generate_numeric_array module allocates and fills the array. The rand() function is used to select an array value which is returned to the random_generator module. The value is converted to an alpha character in the random_generator module and printed in the applicable field in the output file.

Control is returned to the main module and the next attribute record is read. "O" is the next field type read by the main module: control and the attribute record structure is passed to the type_alphanumeric module.

Unlike the other control modules the type_alphanumeric module does not call other modules. The attribute record structure is read to determine the total number of alpha and numeric characters required. Total field width is the aggregate of the two character strings.

The characters are generated sequentially "A - Z" for the alpha string and "0 - 9" for the numeric string. The characters are printed to the output file one at a time until the field is completed.

The process for the other field information types is similar for both the type_alpha and type_numeric modules. ERROR handling is limited to verification of the input data and the opening of the required input and output files.

F. TESTING

Testing was conducted on each module when it was created or updated. Small text files which simulated the input data for the particular module being tested was modified to test each module over a wide range of values. The entire program was tested using a variety of control files to create text files from 50 to over 60,000 records with at least 10 attribute fields per record.

IV. PERFORMANCE ANALYSIS

The purpose of this chapter is to describe and report the results of the empirical study conducted on the three view materialization strategies -- query modification, full materialization and semi-materialization -- using select-project-join (Model 1) and general expressions (Model 2). Performance testing was conducted on databases stored in Random Access Memory (RAM) and on a hard disk using a computer with an INTEL 80386SX processor running at 20 MHz. The simulation program is written in ANSI C with embedded SQL commands to access the INGRES relational database system.

A. SUMMARY OF THE RESULTS FOR THE ANALYTICAL MODEL

Review of the results for the analytical model indicate that view processing strategies are most sensitive to the frequency of updates (P), the selectivity of the view predicate (fv), the selectivity of the query predicate (fq) and number of tuples (l) [Ref. 2]. For select-project-join expressions, and except for high values of P, both full and semi-materialization performed better than query modification.

Higher values of P, fv,l or lower values of fq favor semi-materialization over full materialization. At lower values of P, fv, and l full materialization is slightly better than semi-materialization as the update costs tend to be low.

For general expressions semi-materialization performed better for all parameter values except for very low values of *P*. The absence of an efficient differential algorithm for performing incremental updates makes the use of general expression an unattractive alternative.

B. EXPERIMENTAL SETUP

The parameter definitions, parameter default values, access paths for the relations, query and view definitions and the profiles of the database relations which were used for the experiment are shown in Figures 4 through 8, respectively.

N	Cardinality of the Relation.
K	Number of update transactions on the base relations
l	Total number of tuples modified by each update transaction
q	Number of times the view is queried
P	Probabilitity that a given operation is an update
fv	Selectivity of the view predicate (fraction of tuples in view)
fq	Selectivity of query predicate (fraction of tuples retreived by the query on the view)

Figure 4: View materialization parameter definitions.

N	5000	k	100
I	25	q	100
P	0.5	fv	0.1
fq	0.1		

Figure 5: Default values for parameters.

Relation(s)	Access path
EMP	Clustered index on join field e_num
POS	Clustered index on level
EMP '	Clustered index on join field e_num
POS '	Clustered index on level

Figure 6: Access paths for relations.

The parameters that were considered for the sensitivity analysis include the following for each model tested:

1. The fraction of updates to the total number of operations (P). This parameter is controlled by varying the number of update transactions on the base relations (k) and the number of times the view is queried (q).

2. The selectivity of the view predicate (*fv*) or the fraction of tuples retrieved in the view with regard to the control relation POS. This fraction is controlled by varying the value v_threshold (view_cut) - that is, the predicates in the view definition.

3. The selectivity of the query predicate (*fq*) or the fraction of tuples retrieved by the query on the view. The fraction (*fq*) is controlled by varying the value q_threshold (query_cut) - that is, the predicate in the query.

4. The number of tuples modified by each transaction (*l*). This parameter is controlled by varying the number of tuples per update generated by the data generation program.

5. The number of records in the base relation(s) (cardinality of the relation).

Performance data was collected for view definitions using select-project-join and general expression predicates with the database stored in RAM and on hard disk.

The database and the data generation program, view materialization simulation program and various Ingres program files were placed in separate sub-directories on the hard disk or in two similar RAM drives (4MB for database and 1.8MB for the other files) to determine if eliminating the hard disk access time (28ms average) would significantly improve the performance of the view processing strategies.

	EXPRESSION
VIEW 1:	CREATE VIEW FULL_VIEW SELECT E_NUM, ENAME, SALARY, KEYNO WHERE e.E_NUM = p.E_NUM AND p.LEVEL >= VIEWCUT
VIEW 2:	CREATE VIEW FULL_VIEW SELECT E_NUM, ENAME, SALARY, KEYNO WHERE EXISTS (SELECT * WHERE e.E_NUM = p.E_NUM p_o.KEYNO = p_l.KEYNO AND p_l.LEVEL >= VIEWCUT)
QUERY	SELECT E_NUM, ENAME, KEYNO WHERE SALARY >= QUERYCUT

LEGEND VIEW 1: SELECT-PROJECT-JOIN VIEW 2: GENERAL EXPRESSION

Figure 7: View definitions and query on the views.

Two types of operations were conducted on the test database, a series of update transactions on the base relations which modified a varying number of tuples per update and queries issued against views.

CARD (POS) = 5000

	E#	S#	LEVEL	KEY#	ACCINFO
VAL	500	3	10	25	5000
SIZE	I2	I2	I1	I2	C86

CARD (EMP) = 500

	E#	D#	ENAME	ADDRESS	SALARY	TITLE	JOBDESC
VAL	500	50	500	500	10	500	500
SIZE	I2	I2	C20	C70	I4	C30	C80

Figure 8: Profile of database relations.

The average elapsed time per query for all updates and queries is used to compute the performance of each strategy.

C. PERFORMANCE ANALYSIS

This section discusses the performance of the three view processing strategies for view definitions which used select-project-join and general expression predicates. These strategies were applied to the test database(s) produced using the EMP, POS and SKILL text files generated by the data generation program. The reporting method will consist of reviewing the results for each parameter used in the sensitivity analysis of the two models.

1. Model 1 : Select-Project-Join

MODEL 1 uses the following view definition with a select-project-join predicate for the three view processing strategies:

$$\Pi_{e.ENUM, e.ENAME, e.SALARY}(p.LEVEL \geq viewcut(EMP \bowtie POS))$$

a. Results for Database in RAM

In this section, the results of the sensitivity analysis for Model 1 for the database in RAM is presented. Figures 9 through 12 show the results for model 1 for the four different parameter values when using the ram disk.

In general, the trends computed for the analytical model were supported by the empirical results for the runs with the database stored in RAM.

The sensitivity analysis for the probability of update parameter shows semi-materialization performs best for values of *P* greater than 0.5 with the database in RAM. Full materialization was the clear winner for values of *P* less than 0.5.

This tradeoff occurs because for values greater than 0.5 the cost of processing queries for full materialization averages .35 seconds while the cost to perform updates averaged .7 seconds. The cost per query for semi-materialization averaged .8 seconds but the cost for updates averaged to only .2 seconds.

As the number of updates increased the update cost for semi-materialization was quartered while the cost for full materialization doubled.

The average cost for query modification was 4.3 second per transaction. Query modification exhibited the same trend as the analytical model.

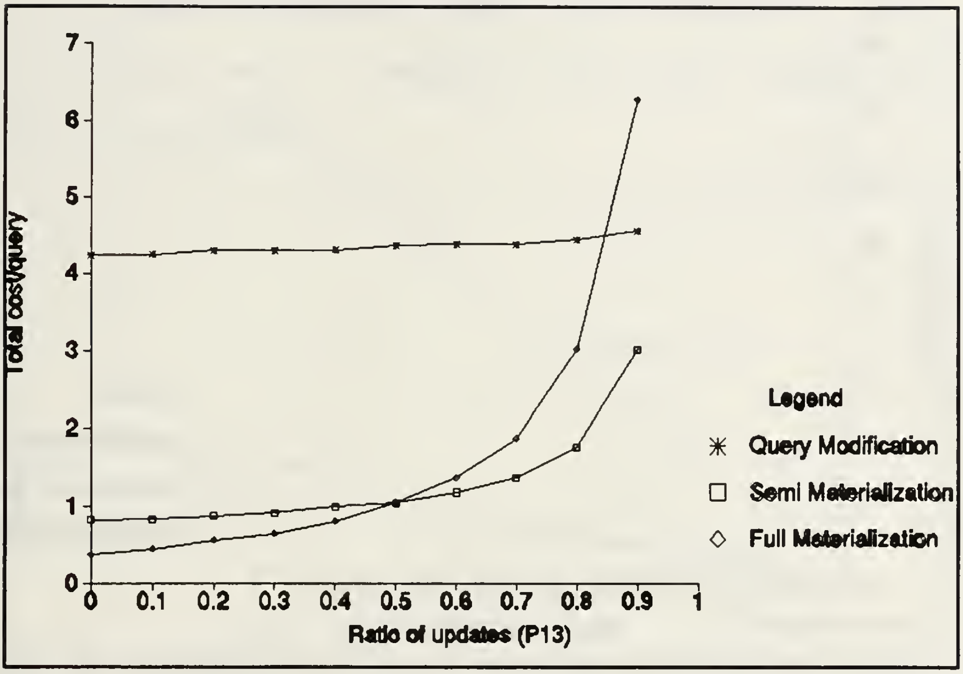

Figure 9: Total cost per query in seconds vs. the ratio of updates to the total number of operations *P*.

For the selectivity of view parameter, the performance of the full and semi-materialization strategies were virtually identical for values of *fv* less than 0.3. The performance of semi-materialization improved over the performance of full materialization as the value of *fv* increased.

The average cost per update was .78 seconds and cost per query was .63 seconds for full materialization for values of *fv* less than 0.3. As expected, the cost for performing updates increased significantly as the value of *fv* increased.

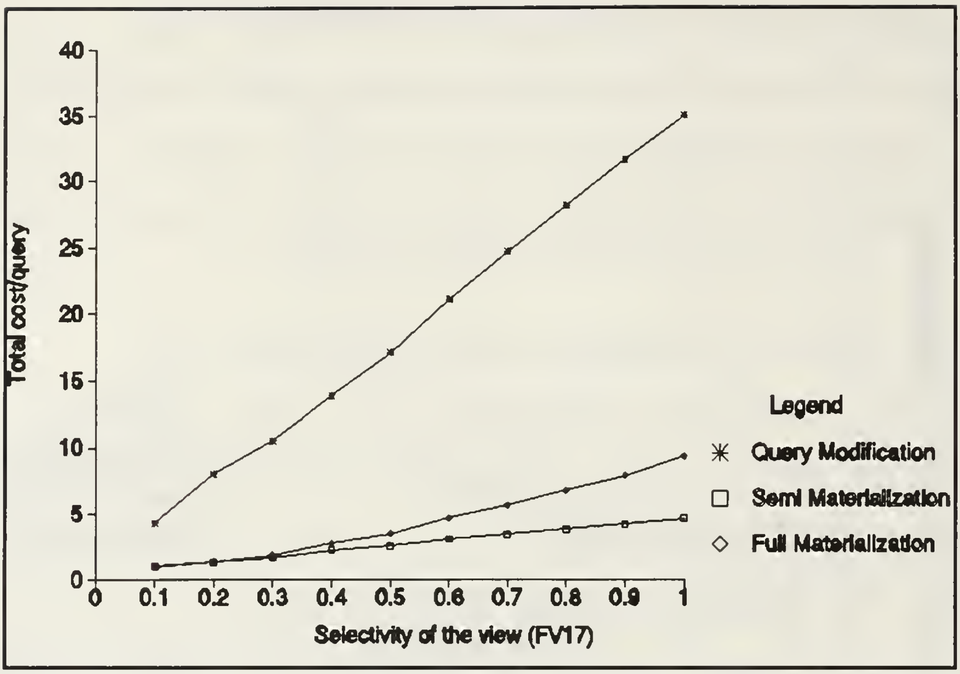

Figure 10: Total cost per query in seconds vs. the selectivity of the view predicate *fv*.

The average cost of updates for semi-materialization was .76 seconds while the cost for queries averaged 3.26 seconds over the entire range of values for *fv*.

The cost for query modification increased as the size of the view increased over the range of *fv* as expected. The empirical results for query modification were virtually identical to the analytical model trends.

Full materialization provided the best performance for all values of *fq* for the strategies with the database in RAM.

The average cost per update for full materialization was .7 seconds while the cost per query averaged 2.2 seconds.

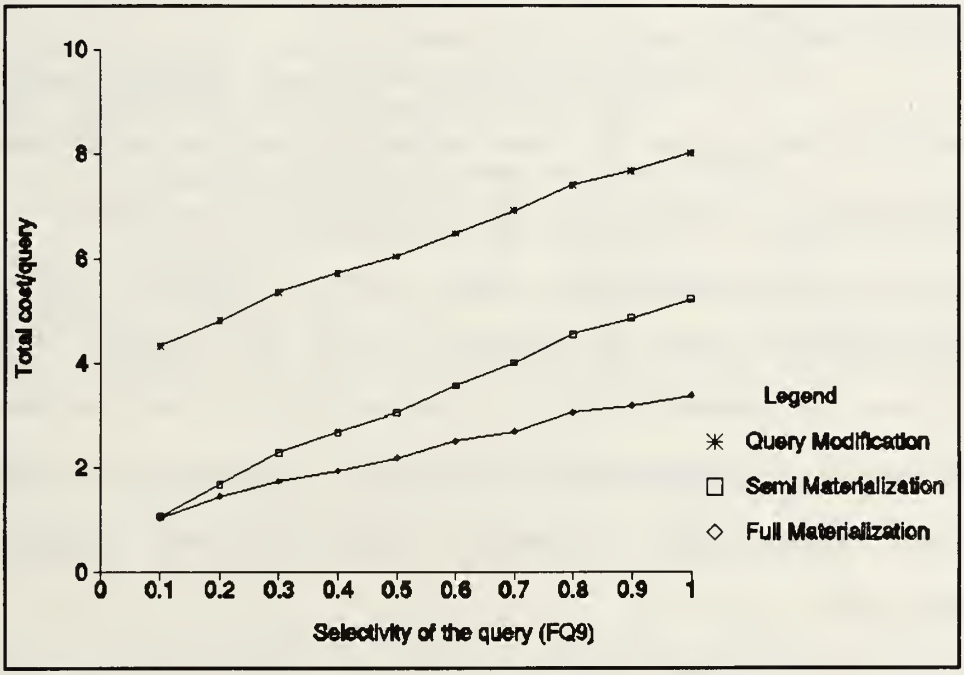

Figure 11: Total cost per query in seconds vs. the selectivity of the query on the view *fq*.

The average cost per update for semi-materialization was .25 seconds but the cost per query averaged 3.7 seconds. Semi-materialization conformed to the trends indicated in the analytical results for *fq*.

The average cost per transaction for query modification was 6.2 seconds which conformed to the performance noted for the analytical model.

There was very little difference between the performances of semi- or full materialization over the entire range of values of *l* - number of tuples per update for the database in RAM.

Full materialization performed slightly better for values of *l* less than 40. Semi-materialization performed best for values of *l* from 50 to 80 and greater than 90.

The average cost per update for semi-materialization was .32 seconds while the average cost per query was .79 seconds for all values of *l*. The average cost per query for full materialization was .36 seconds and average cost per update was .87 seconds over the same range of *l* values.

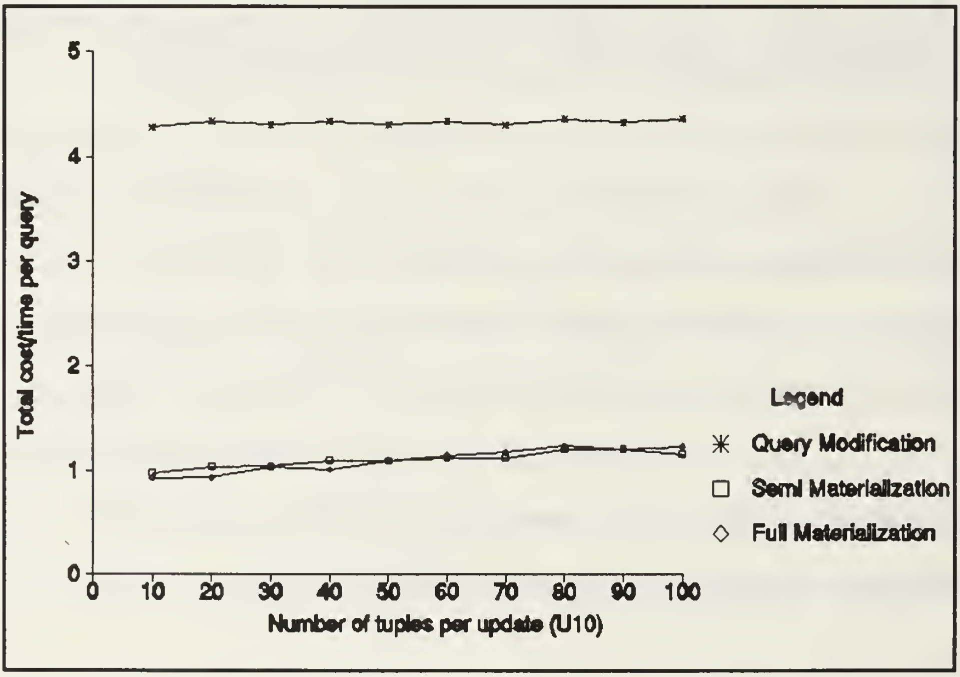

Figure 12: Total cost per query in seconds vs. the number of tuples modified by each update *l*.

Full materialization's performance for the *l* parameter far exceeded the expectations indicated by the analytical model. Query modification's performance on the analytical model indicated a slight improvement for higher values of *l* which was not supported by the empirical data.

b. Results for Database on Hard Disk

In this section the results of the sensitivity analysis for Model 1 on hard disk are presented for comparison to the results for the three strategies with the database in RAM.

For the probability of update parameter, the transition to semi-materialization's performance exceeding full materialization occurs at 0.34. This indicates, as expected, that disk access time when added to the cost of performing updates has an significant impact on the performance of full materialization.

Semi-materialization provides the best performance for values of *P* greater than 0.34. The figure shows a less steep increase for values of *P* greater than 0.7 than the results with the database in RAM. Note, however that the processing cost in RAM is less than 50% of the similar cost on the hard disk.

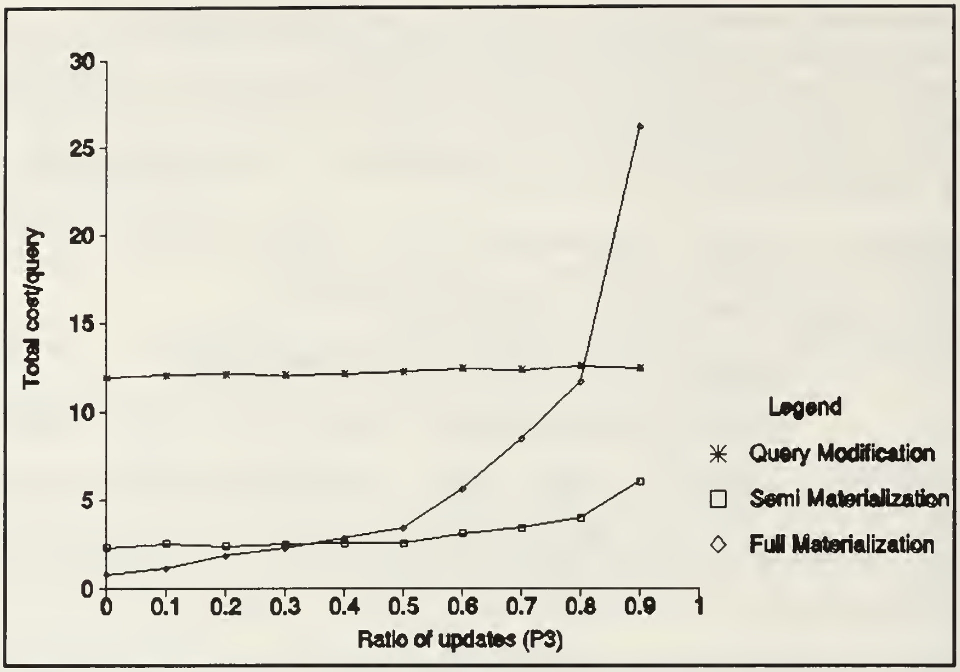

Figure 13: Total cost per query in seconds vs. the ratio of updates to the total number of operations *P*.

For the selectivity on the view parameter, semi-materialization performed best for all values of *fv*. The difference in the performance appears to be due to the additional cost added for accessing the disk to update the base relation plus the additional disk accesses required to update the view.

In general the trends are identical to the trends exhibited by the analytical and RAM models.

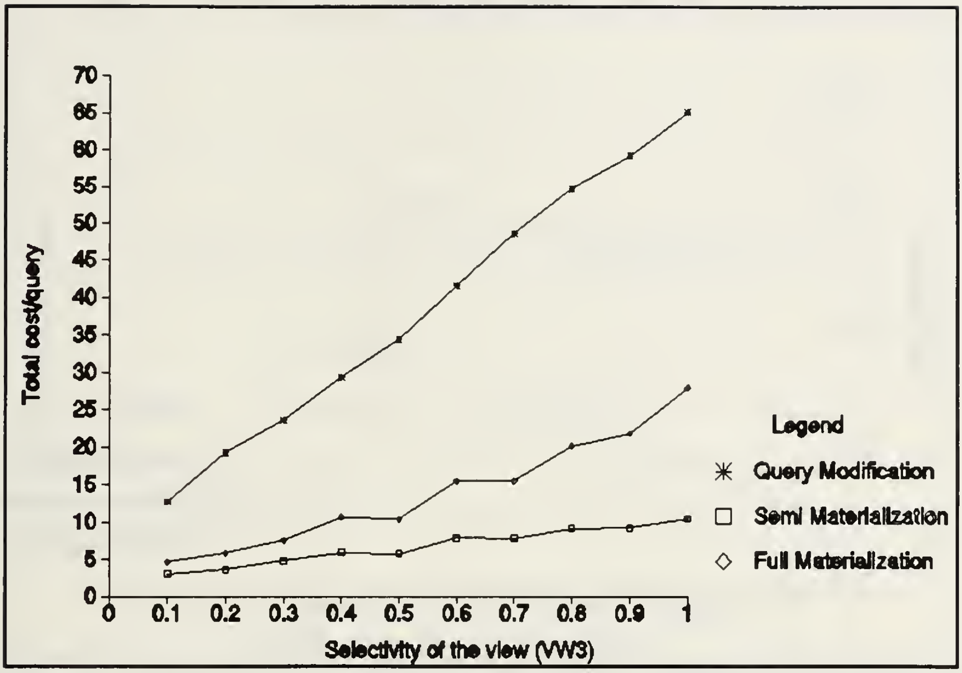

Figure 14: Total cost per query in seconds vs. the selectivity of the view predicate *fv*.

Semi-materialization was the best performer for values of *fq* that were less than 0.4. Full materialization performance was better for values greater than 0.4. The performance for both semi- and full materialization was virtually identical for values of *fq* between 0.2 and 0.4.

The trends for the three strategies conform to the results shown for the analytical model.

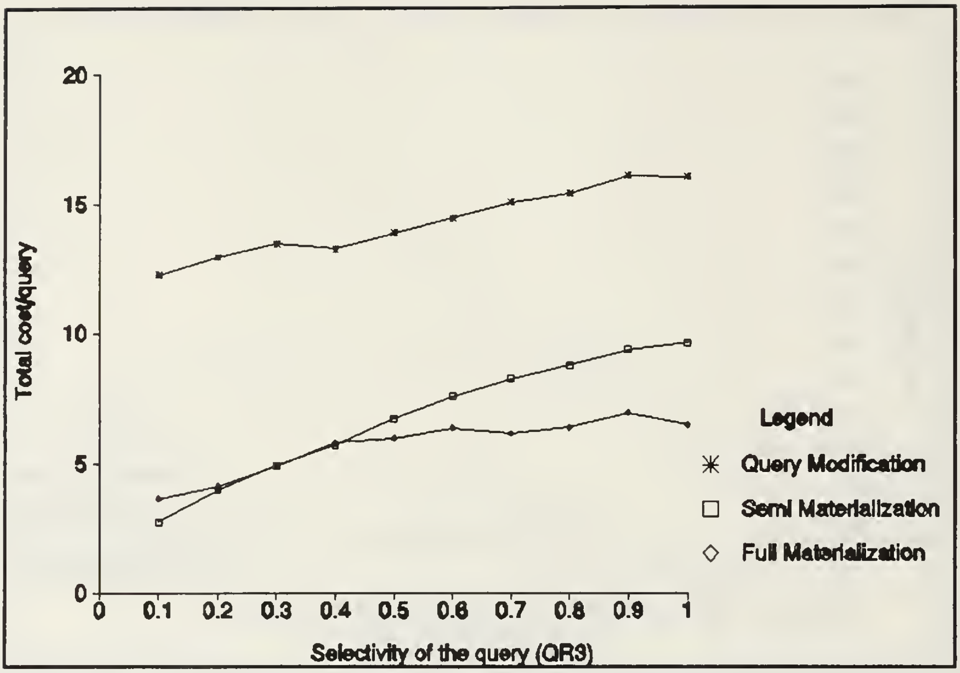

Figure 15: Total cost per query in seconds vs. the selectivity of the query on the view *fq*.

Semi-materialization was the clear winner for all values of *l*. Full materialization 's performance improved for values of *l* greater than 80 but did not out perform semi-materialization. Query modification's performance conformed to both the analytical and RAM models. The results for both semi- and full materialization exceeded the results shown for the analytical model.

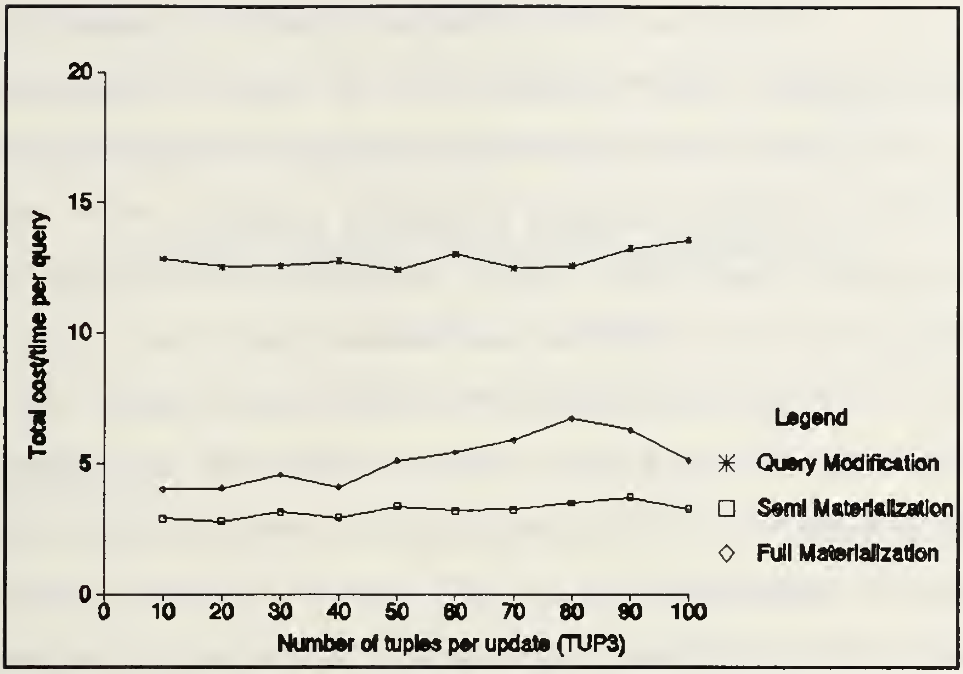

Figure 16: Total cost per query in seconds vs. the number of tuples modified by each update *l*.

c. Discussion of the results for Model 1

In general the empirical data supported the conclusions presented in the analytical review of the view materialization strategies [Ref. 4].

Semi-materialization's performance was superior for higher values of *P*, lower values of *l*, *fv* and all values of *fq* for the database on the hard disk.[1] Semi-materialization performed best with the database in RAM for values of *P* greater than 0.5, *fv* greater than 0.3 and for *l* values between 50 to 80 and greater than 90.

---

[1] Semi-materialization was outperformed by full materialization for only the first value of *fv* while using the RAM disk drive.

This was due to the low cost per update for semi-materialization when compared to the other strategies. The cost advantages of performing queries and updates on the redundant subsets is due primarily to the fact that any transactions performed using semi-materialization are on smaller table(s) than the base relations.

Full materialization performed best for lower values of *P*, *l* and for all values of *fq* for the database on RAM. As expected full materialization performed best when the primary transaction was a query. Surprisingly, full materialization overall performance on RAM was quite good even for the parameters for which it was not the best performer. For example, in Figure 12, full materialization's performance was nearly identical to semi-materialization over the entire range of values for *l*. Similarly, full materialization's performance was not significantly worst than semi-materialization for values of *fv* as shown in Figure 10. Full materialization performed best with the database on hard disk for *P* less than 0.34 and for *fq* values greater than 0.4.

Query modification outperformed full materialization for values of *P* greater than 0.82 for RAM and 0.84 on the hard disk. This was due to the very high cost per update for full materialization as discussed previously.

2. Model 2 : General Expressions

MODEL 2 uses the following general expression view definition expressed in relational calculus:

$$e.ENUM, e.ENAME, e.SALARY \text{ where } \exists (e.ENUM = p.ENUM \wedge p.LEVEL \geq viewcut(EMP \bowtie POS))$$

a. Results for Database in RAM

The results for the database in RAM are displayed in Figures 17 through 20. The trends exhibited here are noteworthy since the performances for all the strategies exceed the results obtained from the earlier experimental data but tended to conform to the analytical model [Ref. 4].

The results for the probability of updates parameter are displayed in Figure 17 and indicate semi-materialization outperformed query modification for all values of *P*. It also outperformed full materialization for values of *P* greater than 0.1.

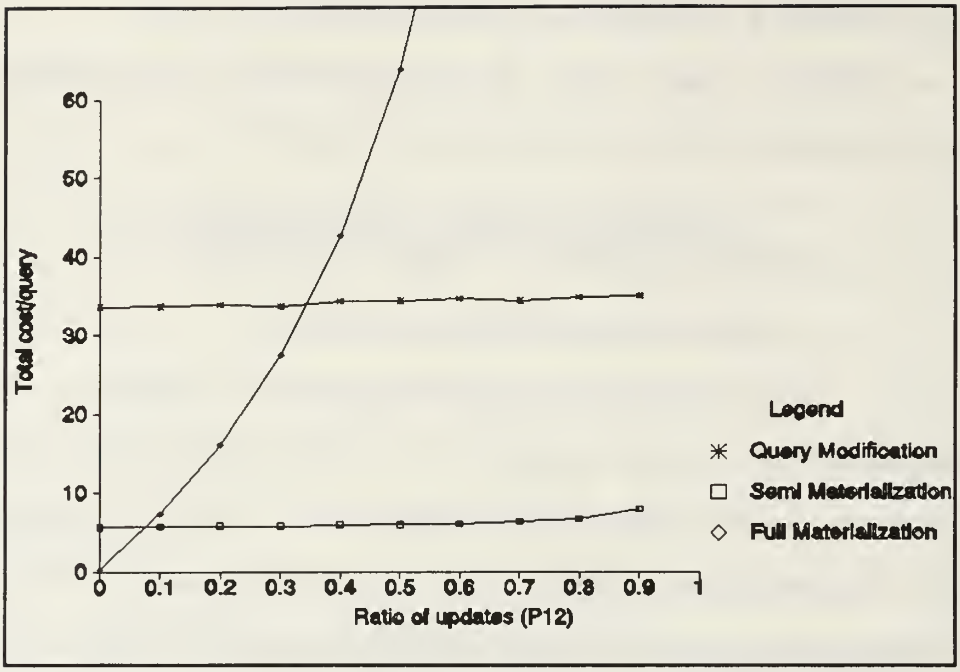

Figure 17: Total cost per query in seconds vs. the ratio of updates to the total number of operations *P*.

Semi-materialization's average cost per update was 0.22 seconds while its cost per query averaged 5.73 seconds over the entire range of *P* values. Full materialization averaged a cost per query of 0.31 seconds but its advantage was offset with an initial update cost of 62.89 seconds. Query modification's cost per transaction was 33.65 seconds.

Semi-materialization was the best performer for all values of *fv*, which coincided with the trend for the analytical model. The simulation results for query modification and full materialization were better than the results for analytical model for all values of *fv*.

Semi-materialization's cost per update averaged 0.68 seconds and its average cost per query was 21.3 seconds over the entire range of *fv* values. Full materialization averaged 1.5 seconds per query and its cost per update averaged 270 seconds. Query modification's cost per transaction averaged to 51.7 seconds for *fv* values.

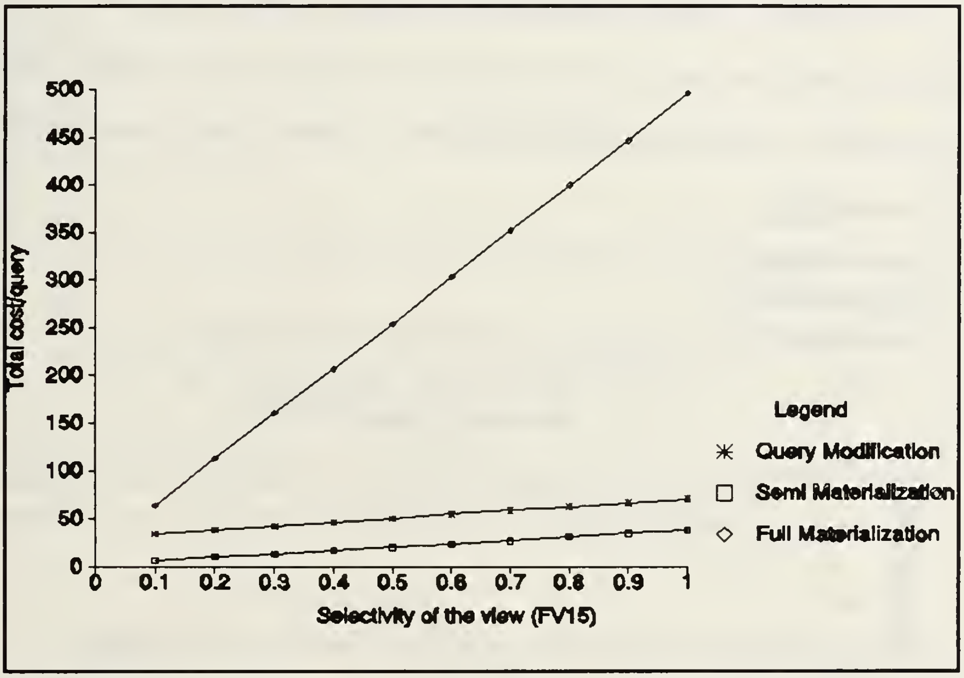

Figure 18: Total cost per query in seconds vs. the selectivity of the view predicate *fv*.

The analysis for the selectivity of the query parameter shows that semi-materialization was the most cost effective strategy for processing queries for general expressions.

The analytical and empirical results for query modification were nearly identical for the entire range of *fq* values. Full materialization proved to be the worst performer of the strategies, as displayed in Figure 19.

Semi-materialization cost per update for *fq* averaged to 0.23 seconds while its average cost per query was 20.7 seconds. The average cost per update for full materialization was 63.5 seconds and its cost per query was 1.3 seconds. Query modification's cost per transaction was 46.1 seconds.

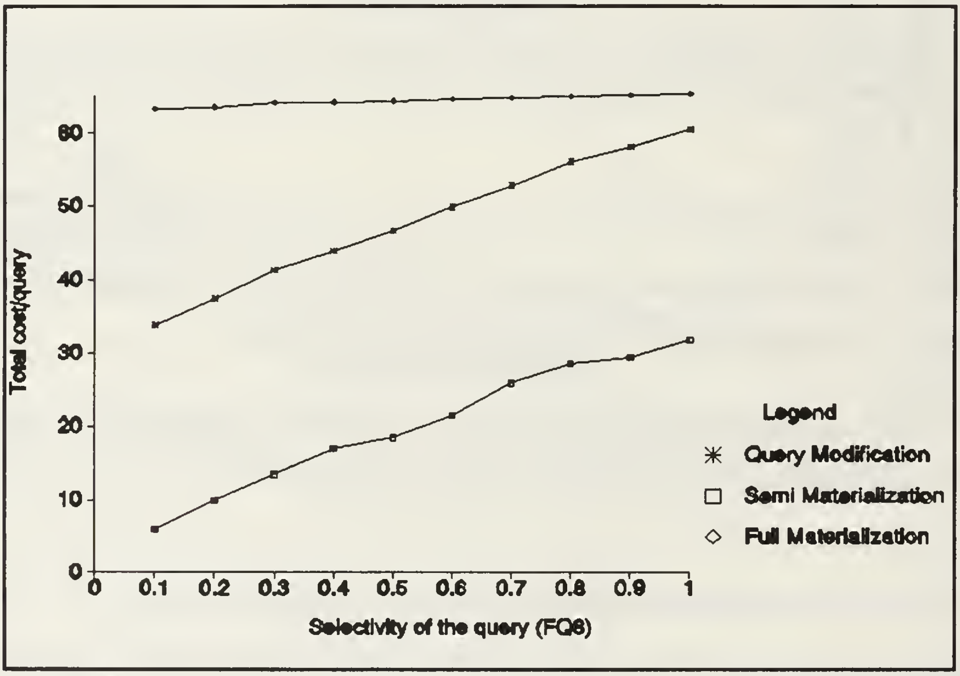

Figure 19: Total cost per query in seconds vs. the selectivity of the query on the view *fq*.

Figure 20 shows that semi-materialization outperformed full materialization and query modification for all values of *l*.

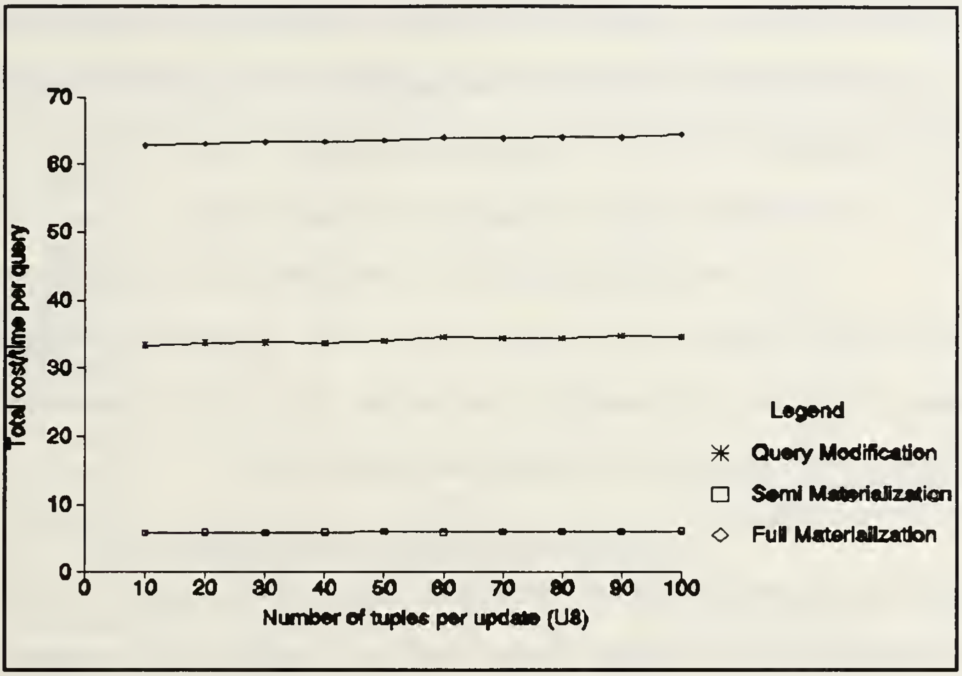

Figure 20: Total cost per query in seconds vs. the number of tuples modified by each update *l*.

Semi-materialization's average cost per update was 0.28 seconds and its cost per query was 5.7 seconds for all values of *l*. Full materialization's cost averaged to 0.32 seconds per query and 63.2 seconds per update. Query modification's transaction cost averaged 33.7 seconds.

The slight increase in performance cost over the entire range of *l* anticipated by the analytical model for semi-materialization as the number of tuples per update increased was not supported by the simulation results. Query modification and full materialization conformed to the results plotted for the analytical model [Ref. 4].

b. Results for Database on Hard Disk

In this section the results of the sensitivity analysis for Model 2 on hard disk are presented. Figures 21 through 28 show the results for model 2 for the five different parameter values when using the hard disk.

Unlike the experiment conducted in RAM for the three view processing strategies, the cardinality of the POS relation will be varied from 7500 to 10,000. The methodology used to conducting the sensitivity analysis for the four other parameter values (*P, fv, fq* and *l*) was exactly the same as the methodology used for the experiment conducted in RAM.

The results for the probability of updates parameter is displayed in Figures 21 and 22 shows that semi-materialization out performed query modification over the entire range of *P* values. Its performance was better than full materialization for values of *P* greater than 0.1.

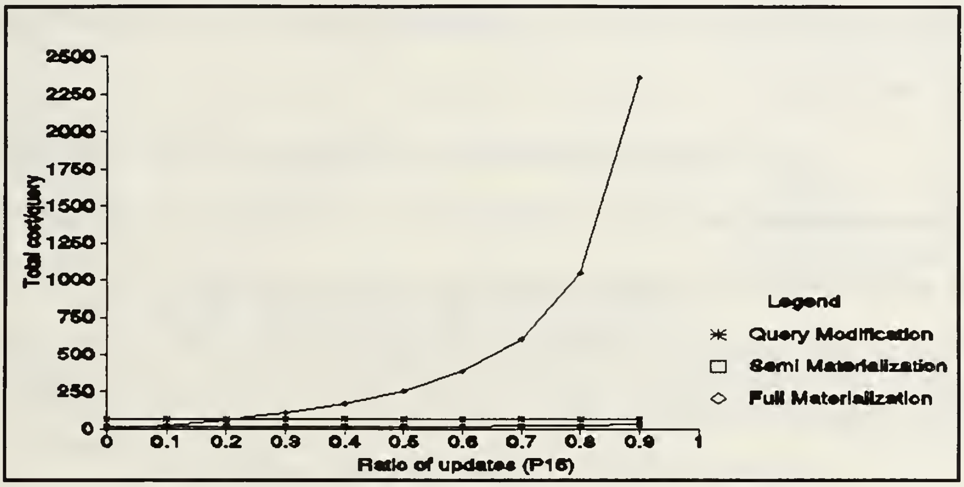

Figure 21: Total cost per query in seconds vs. the ratio of updates to the total number of operations *P* for 7500 records.

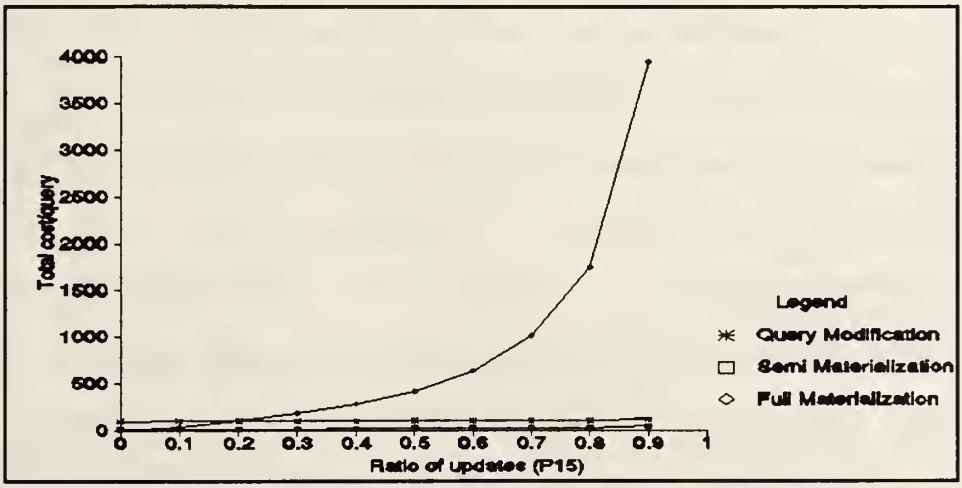

Figure 22: Total cost per query in seconds vs. the ratio of updates to the total number of operations *P* for 10,000 records.

The reason for semi-materialization performance is its low cost per update which offset its average cost per query. For an cardinality of 7500 records, semi-materialization's average cost per update was 1.55 seconds and 19.82 seconds per query. Its average per update for an 10,000 record cardinality was 3.41 seconds with a cost per query of 21.21 seconds.

Full materialization performed best for a *P* value of 0.1 or less for both cardinality values but the extremely high cost of its first update (250 seconds for 7500: 417 seconds for 10,000) quickly overcame its cost advantage for processing queries.

Query modification outperformed full materialization for *P* greater than 0.2 because of full materialization high cost per update.

Semi-materialization outperformed both query modification and full materialization over the entire range of *fv* values. The cost per update for semi-materialization with a cardinality of 7500 was 3.7 seconds and its cost per query was 51.2 seconds for all values of *fv*. Semi-materialization's cost per update for a cardinality of 10,000 records was 7.1 seconds while its cost per query was 59.0 seconds. Query modification cost per transaction averaged 102 seconds for 7500 records and 180 seconds for 10,000 records.

Full materialization cost per update increased at a rate of 150% for 7500 records and doubled for 10,000 records as the size of the view increased.

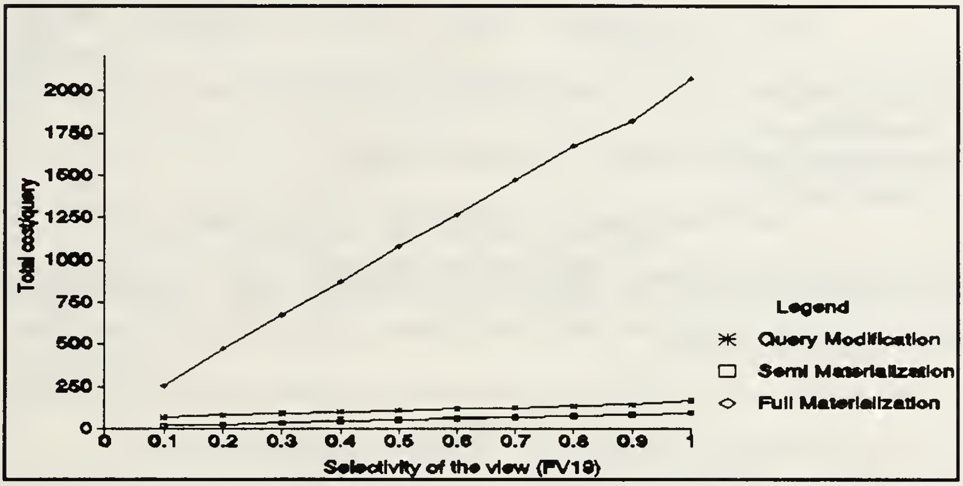

Figure 23: Total cost per query in seconds vs. the selectivity of the view predicate *fv* on 7500 records.

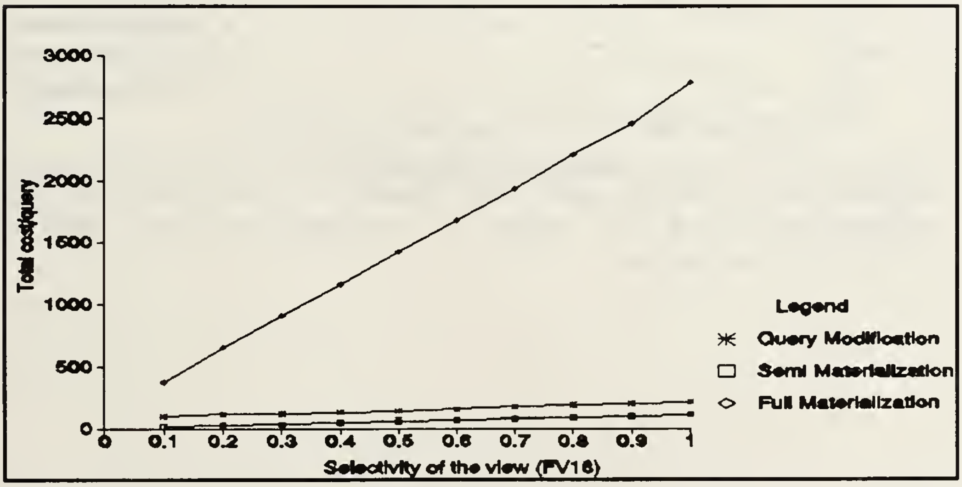

Figure 24: Total cost per query in seconds vs. the selectivity of the view predicate *fv* for 10,000.

As shown in Figures 25 and 26, semi-materialization outperformed query modification and full materialization for all values of *fq*.

The average cost per update for semi-materialization with a cardinality of 7500 was 1.5 seconds and its cost per query was 64.5 seconds.

Semi-materialization's averaged costs for a cardinality of 10,000 was 2.8 seconds for updates and 73.0 seconds for queries.

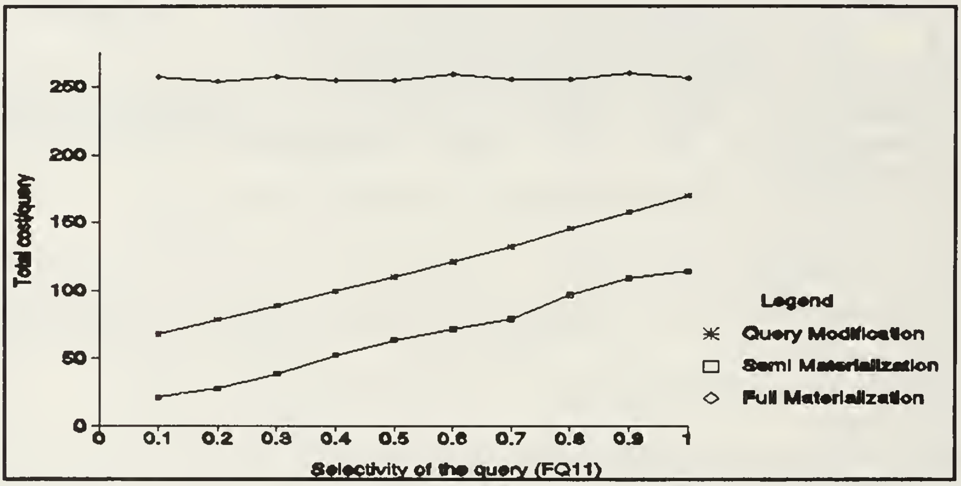

Figure 25: Total cost per query in seconds vs. the selectivity of the query in seconds on the view *fq* for 7500 records.

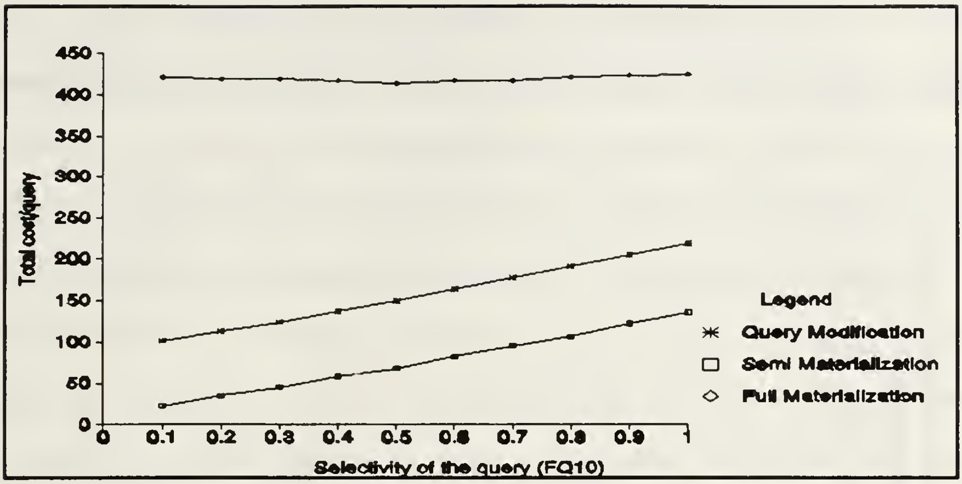

Figure 26: Total cost per query in seconds vs. the selectivity of the query on the view *fq* for 10,000 records.

The averaged costs for full materialization for 7500 records was 2.5 seconds per query and 256 seconds per update.Query modification's costs averaged 116.0 seconds for 7500 records and 150.6 for 10,000 records.

As expected semi-materialization was the clear winner over both query modification and full materialization over the entire range of *l* values.

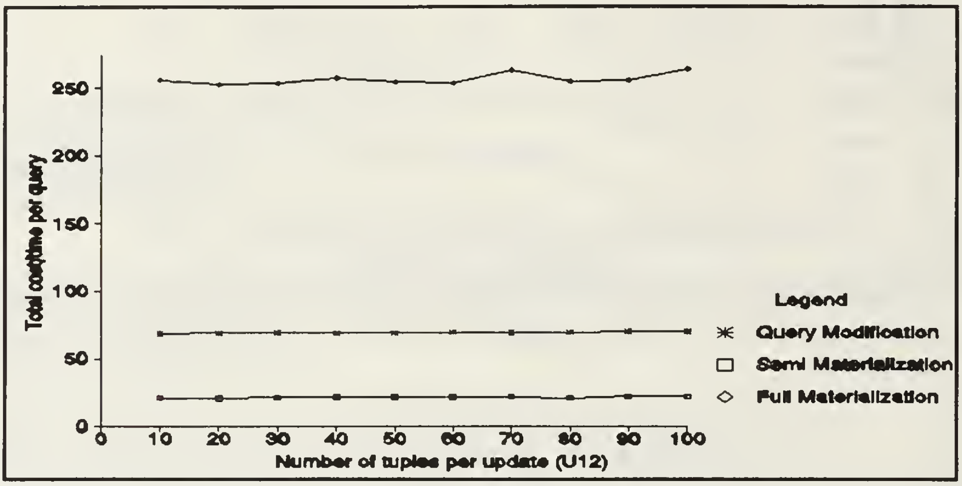

Figure 27: Total cost per query in seconds vs. the number of tuples modified by each update *l* for 7500 records.

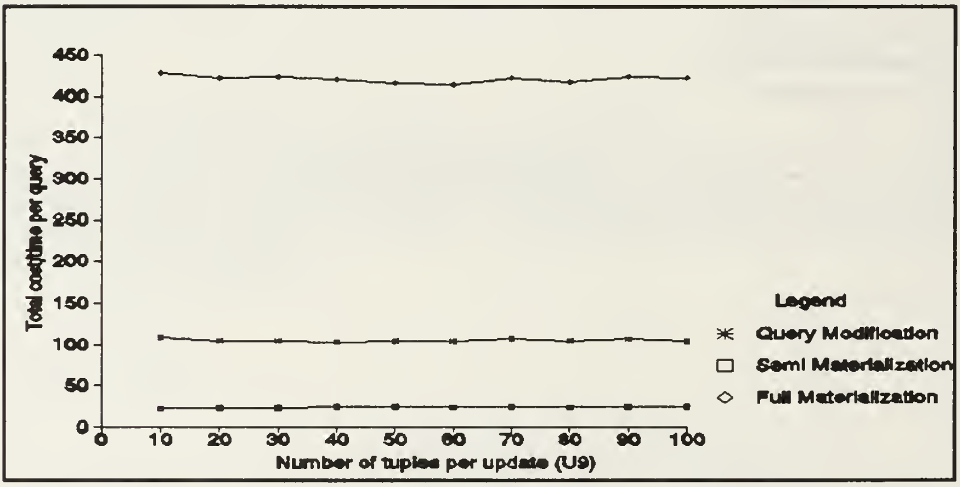

Figure 28: Total cost per query in seconds vs. the number of tuples modified by each update *l* for 10, 000 records.

Semi-materialization's performance cost averaged 2.1 seconds per update and 18.8 seconds per query with a cardinality of 7500 for all values of *l*.

Similarly its averaged costs for 10,000 records were 3.7 seconds per update and 20.2 seconds per query. Full materialization's costs averaged 0.71 per query and 253 seconds per update with a cardinality of 7500. Its average costs were 0.72 seconds per query and 421 seconds per update for 10,000 records. Query modification had an average cost of 68.7 seconds per transaction for 7500 records and 103.2 seconds for 10,000 records.

The additional cost associated with using a hard disk drive and increasing the cardinality had an impact on the performance experienced for both query modification and full materialization.

The cost for processing queries using query modification on the hard disk increased three fold over the cost of processing the query in RAM.

The cost of processing updates using full materialization increased by 700% over the same cost for updates in RAM but the cost for query processing only doubled.

The costs for semi-materialization also increased by a factor of ten for updates and quadrupled for query processing. The increase in processing costs for semi-materialization was offset by the use of the redundant subsets of the base relation which allowed for a more efficient construction of the materialized view.

c. *Discussion of Results for Model 2*

Semi-materialization performed better with the database in RAM and on hard disk for the entire range of values of cardinality than both query modification and full materialization for all parameters except for a value of *P* less than 0.1. Full materialization performed better for *P* less than 0.1 because its average cost per query was only .42 seconds while semi-materialization's cost averaged 15.4 seconds.

For values of *P* greater than 0.1 the cost for performing a single update for full materialization rose to 62.89 seconds for 5000 records in RAM, 250 seconds for 7500 records on hard disk,and 417 seconds for 10,000 records on hard disk which offset any advantage offered by full materialization superior performance for query processing.

This dramatic cost increase for update processing for full materialization using general expression predicates is due the lack of an efficient differential update algorithm. This necessities the complete re-evaluation of the view definition for any update transaction.

Query modification outperformed full materialization for all parameters except for values of *P* less than 0.35 for all values of cardinality. As indicated above the cost of a single update transaction for full materialization quickly drives its cost higher than an other strategies.

In this case as the number of updates increased the aggregate cost for query modification dropped since the cost of updating base relations to support this strategy are not timed. The cost of processing four or less updates (average cost per update for query modification is 0) combined aggregate cost for processing six queries (average cost per query is 33.6 seconds) is more than the cost for processing the same number of transactions for full materialization.

Full materialization's superior performance for lower values of *P* is evident and is based its low cost for processing queries on the view. This advantage was quickly overwhelmed by the overhead of maintaining the fully materialized view [Ref. 4].

Semi-materialization performed best on hard disk for all parameters expect for a value of *P* of 0.1 or less. Full materialization was the best performer for that value of *P* because the only transaction performed for those values was query processing.

As indicated above, semi-materialization is the best strategy for processing view definitions using general expressions for predicates.

It is interesting to note that the results shown in Figures 17 through 28 show that, in general, the performance trends for the view processing strategies are the same for *P*, *fv*, *fq*, and *l* for the entire range of values for cardinality of POS.

V. CONCLUSIONS AND RECOMMENDATIONS

The purpose of this chapter is to state conclusions based on the research and make recommendations for improvements and further study on the three view materialization strategies.

A. CONCLUSIONS

The empirical data of this thesis confirms that the semi-materialization strategy is best method for processing views with predicates using general expressions.

The performance of semi-materialization with the database in RAM exceeded the trends forecasted for general expressions for the analytical model or actual results achieved on the earlier experimental study [Ref. ?]. This is reasonable because of the cost penalty paid by the strategy when it becomes necessary to access a hard disk to perform updates and queries.

The trends for the simulations for semi-materialization with its database stored in RAM indicate it may be suitable for near real time view processing using general expressions based on its relatively low average costs for updates and queries.

Full materialization performed well for lower values of *P* due to its low average cost per query while query modification's performance was good over all parameter values but both strategies are less efficient than semi-materialization for general expressions.

Select-project-join view definitions with the database in RAM proved to be the most cost effective method for view processing (see Figures 9-12).

Overall performance for all parameter simulations using this view definition and the ram disk drive were three to five times faster than similar runs using a hard disk drive. These savings are significant when considering view processing for the small databases inherent to tactical environments.

B. RECOMMENDATIONS AND FUTURE RESEARCH

We recommend that the same simulations for select-project-join and general expressions be conducted with an 80486 processor with a minimum of 16 MB of RAM to test databases with up to 20K records. The internal 8k cache and math co-processor should significantly reduce the processing times for all three strategies using either view definition.

We predict that this approach could improve the performance of the semi-materialization strategy well enough to make it feasible for use with real time tactical systems.

For example, the electronic order of battle maintained on board a tactical aircraft could be completely updated during or enroute to an engagement using information received by its own sensors or sensor information passed from other sources.

This strategy could be used in conjunction with the Joint Ocean Tactical Surveillance (JOTS) system to provide a real time computer generated picture of the tactical and strategic environments.

Used in this manner, JOTS could be placed on board classes of ships which do not have the Naval Tactical Data System (NTDS) installed on board at a tremendous savings over back fitting the vessels with NTDS. The information would improve the vessel's mission performance by keeping the Commanding Officer constantly updated with real time battle group position data and allow for the information received by that vessel's sensors to be incorporated into the tactical picture.

We also recommend conducting more simulations on actual databases with more than two relations, and updates applied to several relations.

Finally, further work is needed to investigate the performance of view processing strategies in the presence of overlapping views over the same relation.

APPENDIX A. DATA GENERATION PROGRAM

```
/*Author:   Curtis Barefield        */
/*Title:    Data Generation Program */
/*version:  MS C 6.0 /C++   (T2)    */
/*created:  17 June 91              */
/*updated:  11 Aug 91                       */

/**********************************************************************************************
*  This program was written to replace the previous hardwired test database                  *
*  generating program with a more generic program. This program generates the text records*
*  used to create the database used to test the view materialization                         *
*  strategies purposed by Professor  Magdi Kamel. The associated test                        *
*  pg has been written by Lt. Jesse  South, USN, a CSM student in class PL03.                *
**********************************************************************************************/

#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <ctype.h>
#include <string.h>
#define size   16  /* sets buffer size for output file name */
#define ALPHA  1 /* set buffer for type/info values */
#define BOUND  6 /* set buffer for upper/lower bounds*/
/*DEBUG TOGGLE */
#define TROUBLESHOOTING           0
/* 1 sends debug data to  output.txt file */
/* prints data used for debugging pg */
void  print_random_generator_array(int, int*), print_bounded_array(int, long*);
/* used to read data into structs */
int rand(), count, increment1, increment2, increment3, t;
long in, results1, results2, results3;
/* modules used to generate random values */
unsigned timer, time_hack();
void srand(unsigned int);
FILE *input_file, *output_file; /* file pointers for text files */
```

```
/**********************************STRUCTS*****************************************************************/
struct       field_attributes
{
      char  field_type[ALPHA];
      int   field_width;
      char  field_information[ALPHA];
      char  lower_bound[BOUND];
      long  increment;
      char  upper_bound[BOUND];
      struct       field_attributes*next;
}ATTRIBUTE;
struct spec_type
{
      long   number_of_records;
      int    number_of_fields;
      char   file_name[size];
      struct field_attributes    *first_field;
      struct spec_type    *next;
}SPECIFICATION;
/* declares modules used to generate attributes */
void type_alpha( struct field_attributes *);
void type_numeric( struct field_attributes *);
void bounded_sequential_array( long, long, int);
void type_alphanumeric( struct field_attributes *);
void counter( int, int);
void random_generator(int, int, int);
void print_database_specifications(struct spec_type *);
void random_long_array(long, long, long);
struct spec_type *list = NULL;
```

```
/*********************************MAIN***************************************/
void   main()
{
       char file_name[size], field_type[ALPHA], field_information[ALPHA];
       char c_lower_bound[BOUND], c_upper_bound[BOUND];
       long number_of_records;
       int  field_width, number_of_fields, i, increment;
      /* creates and defines ptrs used for linked lists */
      struct  field_attributes     *new_attribute= NULL;
      struct  field_attributes     *next_field = NULL;
      struct  field_attributes     *end = NULL;
      struct        spec_type      *new_spec= NULL;
      struct        spec_type      *top_spec  = NULL;
      struct  spec_type    *next_spec= NULL;
      list = NULL;
     input_file= fopen("data_in","r");
     if((input_file)==NULL){
      printf("CAN NOT OPEN INPUT FILE\n");
     }              /*opn input file*/
     /* loads structs and creates linked lists */
     while(!(feof(input_file)))
     {
     /* allocates memory for each specification struct */
     new_spec = (struct spec_type *)malloc(sizeof(struct spec_type));
     if(top_spec == NULL)  /* sets ptr to top of linked list */
     {
      top_spec = new_spec;
      next_spec = new_spec;
      list      = new_spec;
     }
     else
     {
      next_spec->next = new_spec;/* sets ptr to next spec */
      next_spec = next_spec->next;
     }
```

```
/* read in data and loads structs */
if(1 != (fscanf(input_file,"%ld",&number_of_records))){
  printf(" UNEXPECTED VALUE. PROGRAM TERMINATED.");
  exit(0);
  }
next_spec->number_of_records=number_of_records;
if(1 != (fscanf(input_file,"%ld",&number_of_fields))){
  printf(" UNEXPECTED VALUE. PROGRAM TERMINATED.");
  exit(0);
  }
next_spec->number_of_fields=number_of_fields;
if(1 != (fscanf(input_file,"%s",file_name))){
  printf(" UNEXPECTED VALUE. PROGRAM TERMINATED.");
  exit(0);
  }
strcpy(next_spec->file_name,file_name);
next_spec->first_field = NULL;
next_spec->next = NULL;
/* create linked list for attributes */
 for(i=0; i<next_spec->number_of_fields; ++i)
 {
   /* allocates memory for attribute struct */
  new_attribute =(struct field_attributes *) malloc (sizeof(struct field_attributes));
  if(next_spec->first_field == NULL)
  {
        next_spec->first_field = new_attribute;    /* sets ptr to top of list */
        end = next_spec->first_field;
  }
  else
  {
        end->next = new_attribute;             /* sets ptr to next attribute */
        end = end->next;
  }
  /* load attribute struct */
  if(1 != (fscanf(input_file,"%s",field_type))){
        printf(" UNEXPECTED VALUE. PROGRAM TERMINATED.");
        exit(0);
        }
```

```
    strcpy(end->field_type,field_type);
    if(1 != (fscanf(input_file,"%d",&field_width))){
        printf(" UNEXPECTED VALUE. PROGRAM TERMINATED.");
        exit(0);
        }
    end->field_width=field_width;
    if(1 != (fscanf(input_file,"%s",field_information))){
        printf(" UNEXPECTED VALUE. PROGRAM TERMINATED.");
        exit(0);
        }
    strcpy(end->field_information,field_information);
    if(1 != (fscanf(input_file,"%s",c_lower_bound))){
        printf(" UNEXPECTED VALUE. PROGRAM TERMINATED.");
        exit(0);
        }
    strcpy(end->lower_bound,c_lower_bound);
    if(1 != (fscanf(input_file,"%d",&increment))){
        printf(" UNEXPECTED VALUE. PROGRAM TERMINATED.");
        exit(0);
        }
    end->increment = increment;
    if(1 != (fscanf(input_file,"%s",c_upper_bound))){
        printf(" UNEXPECTED VALUE. PROGRAM TERMINATED.");
        exit(0);
        }
    strcpy(end->upper_bound,c_upper_bound);
    end ->next =NULL;
   }
  }
    fclose(input_file);
    /* Redirects monitor output to text file called output.txt */
    #if  TROUBLESHOOTING
        freopen("output.txt","w",stdout);
        print_database_specifications(list);
        t=0;
        #define DEEP_DEBUG   1
    #endif
```

```
/* uses system time to generate random number */
  timer=time_hack();
  srand(timer);
  /* uses linked list to read each spec (BEGIN PG EXEC ) */
  while(list != NULL)
  {
       results1=0; /* sets module "counter" to zero */
       results2=0;
       results3=0;
       increment1=0;
       increment2=0;
       increment3=0;
       output_file= fopen(list->file_name,"w");
       if((output_file)==NULL){
             printf("CAN NOT OPEN OUTPUT FILE\n");
             exit(0);
       }            /*opn output file*/
             /* BUILD DATABASE TEXT FILES */
       for(in=0;in<list->number_of_records;++in)
       {
             next_field = list->first_field;
             count=0;
             while(next_field !=NULL)
             {
                   if(next_field -> field_type[0]=='A')
                   {
                         type_alpha( next_field);
                   }
                   else if(next_field -> field_type[0]=='N')
                   {
                         type_numeric( next_field);
                   }
                   else if(next_field -> field_type[0]=='O')
                   {
                         type_alphanumeric(next_field);
                   }
```

```
                next_field = next_field->next;
                if(next_field !=NULL)(
                      fprintf(output_file,":");
                      }
                }
                fprintf(output_file,"\n");
          }
          list = list->next;
          fclose(output_file);
      }
      exit(0);
}
/***********************************************************************
*      Time Hack uses the system clock to provide the seed value    *
*      for the rand and srand library functions.    *
***********************************************************************/
unsigned time_hack()
{
    clock_t   rand_input;
    unsigned seed_input;
     rand_input=clock();
     seed_input=(unsigned)rand_input;
     return(seed_input);
}
/************************************************************************
*                 Bounded sequential array                       *
*      creates a bounded array which is used with the rand() library  *
*      function to select from a user specified number of array       *
*       elements to return a value from within that array.          *
************************************************************************/
void bounded_sequential_array(long_increment, long_lower_bound, int number_of_values )
{
    int m, long_index;
    long  *long_storage,long_low;
    /* allocate storage space in memory for array */
    long_storage=(long*)calloc(number_of_values,sizeof(long));
```

```
    /* fill array */
    long_storage[0]=long_low=long_lower_bound;
    for(m=1; m< number_of_values; m++)
    {
     long_storage(m]=long_low +=long_increment;
    }
    /* select value fm array using generic random generator */
    long_index=(rand()%number_of_values);
    fprintf(output_file,"%05ld",long_storage[long_index]);
    #if DEEP_DEBUG
       if(t<1){
       print_bounded_array(number_of_values, long_storage);}
       t++;
    #endif
}
/*****************************************************************************
*        Type alpha generates alpha character string to fill user           *
*     determined field size                                                *
*****************************************************************************/
void type_alpha( struct field_attributes *next_field)
{
    char c;
    static char start='A'-1;
    int j, increment, lower_bound, upper_bound;
    int number_of_values;
    long_increment, long_lower_bound;
    /******* Conversions for Random Generator ************/
     lower_bound=(int)(next_field ->lower_bound[0]);
     upper_bound=(int)(next_field ->upper_bound[0]);
     increment=next_field ->increment;
     /************** Conversions for Bounded Array ********/
     long_lower_bound=(atol)(next_field ->lower_bound);
     long_increment=next_field ->increment;
     number_of_values=(atoi)(next_field ->upper_bound);
     if(next_field ->field_information[0]=='R') /* Random field selected */
     {
          random_generator(increment, lower_bound, upper_bound);
     }
```

```
      else if(next_field ->field_information[0]=='B') /* Bounded sequential  field
        selected */
      {
            bounded_sequential_array(long_increment, long_lower_bound, number_of_values);
      }
      else if(next_field ->field_information[0]=='X'||'D')
        /* countup counter places chars 'A'-'Z' in field */
      {
            for(j=0;j<next_field ->field_width;++j)  /* sets field width */
            {
               if(start >='Z')
               {
                      start= 'A';
               }
               else
               {
                      ++start;
               }
               fprintf(output_file,"%c",start); /* prints alpha char to
                             output file */
            }
      }
}
/*******************************************************************
*       Type numeic generates numeric chararters to fill user   *
*       determined field size                                   *
*******************************************************************/
void type_numeric( struct field_attributes *next_field)
{
    char c;
    static start='0'-1;
    int j, lower_bound, increment, upper_bound;
    int number_of_values, increment_c, lower_bound_c;
    long_increment, long_lower_bound, increment_long_array;
    long lower_bound_long_array, upper_bound_long_array;
```

```
/************** Conversions for Counter ***************/
increment_c = next_field->increment;
lower_bound_c =(atoi)(next_field->lower_bound);
/******* Conversions for Random Generator ************/
 lower_bound=(int)(next_field ->lower_bound[0]);
 upper_bound=(int)(next_field ->upper_bound[0]);
 increment=next_field ->increment;
 /************** Conversions for Bounded Array ********/
 long_lower_bound=(atol)(next_field ->lower_bound);
 long_increment=next_field ->increment;
 number_of_values=(atoi)(next_field ->upper_bound);
 /************** Conversions for Random Long Array ********/
 lower_bound_long_array =(atol)(next_field ->lower_bound);
 increment_long_array   =(long)(next_field ->increment);
 upper_bound_long_array =(atol)(next_field ->upper_bound);
if(next_field ->field_information[0]=='S')   /* sequential counter */
{
   counter(increment_c, lower_bound_c);
   count++;
}
else if(next_field ->field_information[0]=='R') /* Random */
{
   random_generator( increment, lower_bound, upper_bound);
}
else if(next_field ->field_information[0]=='B') /* Bounded sequential */
{
   bounded_sequential_array(long_increment, long_lower_bound, number_of_values );
}
else if(next_field ->field_information[0]=='X') /* Random long array */
{
   random_long_array(increment_long_array, lower_bound_long_array, upper_bound_long_array );
}
else if(next_field ->field_information[0]=='D')
```

```
        /* countup counter, fills field with '0' - '9' in sequence */
    {
      for(j=0;j<next_field ->field_width;++j)  /* sets field width */
      {
            if(start >='9')
            {
                  start= '0';
            }
            else
            {
                  ++start;
            }
            fprintf(output_file,"%c",start); /* prints numeric char to
                        output file */
      }
     }
}
/***********************************************************************
*       Type alphanumeric generates alphanumeric chararters to fill    *
*       user determined field size                                     *
***********************************************************************/
void type_alphanumeric(struct field_attributes *next_field)
{
    char c, d;
    int  j, k  /*, count=0*/ ;
    static char alpha='A'-1;
    static char numeric='0'-1;
    int alpha_field_width, numeric_field_width;
    alpha_field_width=(atoi)(next_field ->lower_bound);
    numeric_field_width=(atoi)(next_field ->upper_bound);
```

```
    for(j=0;j<alpha_field_width;++j)   /* generates  alpha chars 'A'-'Z' */
    {
       if(alpha >='Z')
       {
         alpha='A';
       }
       else
       {
          ++alpha;
       }
       fprintf(output_file,"%c",alpha);
    }
    for(k=0;k<numeric_field_width;++k)   /* numerics '0'-'9' */
    {
       if(numeric >='9')
       {
         numeric='0';
       }
       else
       {
         ++numeric;
       }
       fprintf(output_file,"%c",numeric);
    }
}
```

```
/***************************************************************************
*             Random generator takes a lower, upper and                  *
*             increment int value, creates an  numeric array  and        *
*             uses the functions rand() and srand() to                   *
*             select an array element which maybe converted to a         *
*             alphanumeric char for printing to the output file.         *
***************************************************************************/
void random_generator( int increment, int lower_bound, int upper_bound)
{
     /* Declare module data elements */
        int  numeric_index;
        int  output_from_array;
        int  generate_numeric_array(int,int,int);
        char alnum_character_output;
     /* Compute array size */
     numeric_index= ((upper_bound-lower_bound)/increment);
     /* Determine if a set of bounded random numbers are required */
     if(((lower_bound>47)&&(upper_bound<58)) ||  (lower_bound>64)&&(upper_bound<91))
     {
          output_from_array= generate_numeric_array(numeric_index, increment, lower_bound);
          alnum_character_output=(char)(output_from_array);
          fprintf(output_file, "%04c",alnum_character_output);
     }
     else
     {
         output_from_array= generate_numeric_array(numeric_index, increment, lower_bound);
         fprintf(output_file, "%04d", output_from_array);
     }
}
```

```
/*************************************************************************
*            Random long array takes a lower, upper and            *
*            increment long value, creates an numeric array  and    *
*            uses the functions rand() and srand() to       *
*            select an array element which is  printed to           *
*            the output file.                                       *
*************************************************************************/
void   random_long_array(   long   increment_long_array,   long   lower_bound_long_array,   long
upper_bound_long_array)
{
     /* Declare module data elements */
        int m, index;
        long numeric_index, low;
        long *output_from_array;
        long output_from_random;
     /* Compute array size */
     numeric_index= ((upper_bound_long_array-lower_bound_long_array)/increment_long_array);
     if(numeric_index<20)
     {
       /* Allocate memory block for long array*/
       output_from_array= (long*)calloc(numeric_index,sizeof(long));
       /* Set lower bound of array */
       output_from_array[0]= low= lower_bound_long_array;
       /* Load array */
       for(m=1;m<numeric_index;m++)
       {
              output_from_array[m]= low+= increment_long_array;
       }
       index= (rand()%numeric_index);
       fprintf(output_file,"%04ld",output_from_array[index]);
     }
     else
     {
       output_from_random= (1+(rand()%upper_bound_long_array));
       fprintf(output_file,"%04ld",output_from_random);
     }
}
```

```
/***************************************************************************
*            Counter uses lower bound and increment to act as             *
*            a sequential counter for max of numbers per spec_type.       *
***************************************************************************/
void counter( int increment_c, int lower_bound_c)
{
       /************** FIRST COUNTER ************/
       if((in==0)&&(results1==0)&&(count==0))
       {
                results1 = lower_bound_c;
                increment1 = increment_c;
       }
       else if(count==0)
       {
            results1 = results1+increment1;
       }
       if(count==0){
             fprintf(output_file,"%041d",results1);}
       /************ SECOND COUNTER *************/
       if((in==0)&&(results2==0)&&(count==1))
       {
                results2 = lower_bound_c;
                increment2 = increment_c;
       }
       else if(count==1)
       {
            results2 = results2+increment2;
       }
       if(count==1){
             fprintf(output_file,"%041d",results2);}
       /************* THIRD COUNTER *************/
       if((in==0)&&(results3==0)&&(count==2))
       {
                results3 = lower_bound_c;
                increment3 = increment_c;
       }
```

```
        else if(count==2)
        {
             results3 = results3+increment3;
        }
        if(count==2){
              fprintf(output_file,"%04ld",results3);}
        /******************************************/
}
/*************************************************************************
*             Generate numeric array produces a bounded                  *
*       array and uses the rand() function to simulate a real           *
*             random number generator                                    *
*************************************************************************/
int generate_numeric_array(int numeric_index, int increment,
        int lower_bound)
{
            int m, index;
            int *numeric_storage, low;
            numeric_storage=(int*)calloc(numeric_index,sizeof(int));
            numeric_storage[0]= low = lower_bound;
            for(m=1; m<numeric_index; m++)
            {
               numeric_storage[m]=low += increment;
            }
            index=(rand()%numeric_index);
            return(numeric_storage[index]);
            #if DEEP_DEBUG
               if(t<1){
               print_random_generator_array(numeric_index, numeric_storage};}
               t++;
           #endif
}
```

```
/************* PRINT RANDOM GENERATOR ARRAY ******************************/
void print_random_generator_array(int numeric_index,int *numeric_storage)
{
	int i;
	for(i=0;i<numeric_index;i++){
		printf(" array[%d] = %d\n",i,numeric_storage[i]);
	}
}
/**************** PRINT BOUNDED ARRAY ************************************/
void print_bounded_array(int number_of_values,long *long_storage)
{
	int i;
	for(i=0;i<number_of_values;i++){
		printf(" array[%d] = %05ld\n",i,long_storage[i]);
	}
}
/*************************************************************************
*	Prints specifications including all attribute link lists	*
*************************************************************************/
void print_database_specifications(struct spec_type *list)
{
	struct field_attributes	*next_field;
	int i, l=0;
	printf("Printing link lists for generic database generator\n");
	while(list !=NULL)
	{
		i=0;

		printf("\tSPEC %d\n",++l);
		printf("\tThere will be %ld records\n",list->number_of_records);
		printf("\Tthere will be %d fields\n",list->number_of_fields);
		printf("\tThe file name is %s\n",list->file_name);
		next_field = list->first_field;
		while(next_field !=NULL)
		{
			printf("\tField %d:\n",++i);
			printf("\t%s is field type\n",next_field->field_type);
			printf("\t%d is field width\n",next_field->field_width);
```

```
                printf("\t%s is field info\n",next_field->field_information);

                printf("\t%s is lower bound\n",next_field->lower_bound);

                printf("\t%ld is increment\n",next_field->increment);

                printf("\t%s is upper bound\n",next_field->upper_bound);

                next_field = next_field->next;

            }

            list = list->next;

        }

}
```

APPENDIX B. VIEW MATERIALIZATION SIMULATION PROGRAM

```
/* Title           : View Materialization Simulation (vsgxpdp7)          */
/* Author          : Jesse T. South                                      */
/* Date            : 17 June 1991                                        */
/* Revised         : 25 July 1991                                        */
/* Modified        : for general expressions 22 AUG by Curtis Barefield  */
/* Purpose         : Thesis Research                                     */
/* System            : IBM 80286 clone/ 80386SX                          */
/* Compiler          : Microsoft C 6.0, INGRES precompiler,(Borland C++) */
/* Description       : The program is part of a thesis                   */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
exec sql include sqlca;
#define size 16
#define dbinfo "info.dat"
#define cntrlfl "cntrl.dat"
#define update_file "data_in"
#define finrslt "fnlrslt.dat"
#define runrslt "rnrslt.dat"
exec sql begin declare section;
#define empinfo "empdat.dat"
#define posinfo "posdat.dat"
#define skilinfo "skildat.dat"
#define updatinfo "update.dat"
exec sql end declare section;
void open_files(FILE**, FILE**, FILE**);
void close_files(FILE**, FILE**, FILE**);
void init_test_database(int);
void scan_dbinfo(long*, long*, long*, int*, int*, int*, long*, long*, long*);
void create_tables(void);
void create_views(int);
void create_update_table(void);
void copy_base_tables(void);
void copy_semi_n_full_mats(int);
void create_table_index(void);
void module_qm(char, int, long, double*, FILE*);
void module_sm(char, int, long, double*, FILE*);
```

```
void module_fm(char, int, long, double*, FILE*);
void write_file_headings(char*, char*, FILE*, FILE*);
void write_run_result(char, char, int, long, double, long, FILE*);
void write_final_result(int, int, long, int, long, long, long, float,
        float, float, float, float, double, double, double, FILE*, FILE*);
void compute_avg_time(int, double*, double*, double*);
void compute_fv_and_fq_and_P(int, int, int, int, float*, long, long, long,
                long, float*, int, int, float*);
void compute_table_counts(long*, long*, long*, long, float*, float*);
void refresh_update_text_file(long, long, long);
void main(void)
  {
  int K, Q, updat_siz, i, run_cnt = 0, zero = 0;
  int vmax, vbase, vincr, viewcut;
  long ecard, pcard, scard, countb, countv, countq;
  long qmax, qbase, qincr, querycut;
  float fv, fva, fq, fqa, P;
  double timeqm, timesm, timefm;
  char QUERY = 'Q', UPDATE = 'K';
  char *prm_ptr, parameter[10], *updt_ptr, updat_rel[10];
  FILE  *cntrl_fl, *fresult_fl, *run_rslt;
  prm_ptr = &parameter[0];
  updt_ptr = &updat_rel[0];
  open_files(&run_rslt, &cntrl_fl, &fresult_fl);
  scan_dbinfo(&pcard, &ecard, &scard, &vmax, &vbase, &vincr, &qmax, &qbase,
          &qincr);
  while(!feof(cntrl_fl))
    {
    timeqm = timesm = timefm = 0.0;
    countb = countv = countq = 0;
    fscanf(cntrl_fl, "%d %ld %d %d %d %s %s", &viewcut, &querycut, &K, &Q,
         &updat_siz, prm_ptr, updt_ptr);
    if (run_cnt == zero) write_file_headings(prm_ptr, updt_ptr, fresult_fl,
                                        run_rslt);
    init_test_database(viewcut);
    run_cnt++;
    printf("\n run # %d\n", run_cnt);
```

```
    for(i = 0; i < K; i++)
      {
      refresh_update_text_file(pcard, i, updat_siz);
      module_qm(UPDATE, viewcut, querycut, &timeqm, run_rslt);
      module_sm(UPDATE, viewcut, querycut, &timesm, run_rslt);
      module_fm(UPDATE, viewcut, querycut, &timefm, run_rslt);
      }
    for(i = 0; i < Q; i++)
      {
      module_qm(QUERY, viewcut, querycut, &timeqm, run_rslt);
      module_sm(QUERY, viewcut, querycut, &timesm, run_rslt);
      module_fm(QUERY, viewcut, querycut, &timefm, run_rslt);
      }
    compute_avg_time(Q, &timeqm, &timesm, &timefm);
    compute_fv_and_fq_and_P(vmax, vbase, vincr, viewcut, &fv, qmax, qbase,
                       qincr, querycut, &fq, K, Q, &P);
    compute_table_counts(&countb, &countv, &countq, querycut, &fva, &fqa);
    write_final_result(run_cnt, viewcut, querycut, updat_siz, countb, countv,
                    countq, fv, fva, fq, fqa, P, timeqm, timesm, timefm,
                    fresult_fl, run_rslt);
    exec sql disconnect;
    system("rmingres");
    }
  close_files(&run_rslt, &cntrl_fl, &fresult_fl);
  printf("\ndisconnect complete\n");
  }
```

```
void init_test_database(int viewcut)
  {
  system("destroydb magdi");
  system("createdb magdi");
  system("addingres -B -D64000");
  exec sql whenever sqlerror stop;
  exec sql connect magdi;
  create_tables();
  create_views(viewcut);
  copy_base_tables();
  copy_semi_n_full_mats(viewcut);
  create_table_index();
  }
void open_files(FILE **run_rslt, FILE **cntrl_fl, FILE **fresult_fl)
  {
  *cntrl_fl = fopen(cntrlfl, "r");
  *fresult_fl = fopen(finrslt, "a");
  *run_rslt = fopen(runrslt, "a");
  if((!*run_rslt) || (!*cntrl_fl) || (!*fresult_fl))
    {
    printf("\nERROR: control or output files did not open");
    fcloseall();
    exec sql disconnect;
    exit(1);
    }
  }
void close_files(FILE **run_rslt, FILE **cntrl_fl, FILE **fresult_fl)
  {
  int i;
  fprintf(*fresult_fl,"\n");
  for (i=0;i<80;i++) fprintf(*fresult_fl,"*");
  fclose (*run_rslt);
  fclose (*cntrl_fl);
  fclose (*fresult_fl);
  }
```

```
void scan_dbinfo(long* ecard, long* pcard, long* scard, int* vmax, int* vbase,
              int* vincr, long* qmax, long* qbase, long* qincr)
  {
  FILE* db_info;
  db_info = fopen(dbinfo, "r");
  if(!db_info)
    {
    printf("\NERROR: dbinfo file did not open ");
    fcloseall();
    exec sql disconnect;
    exit(1);
    }
  fscanf(db_info, "%ld %ld %ld\n", &*ecard, &*pcard, &*scard);
  fscanf(db_info, "%d %d %d\n", &*vmax, &*vbase, &*vincr);
  fscanf(db_info, "%ld %ld %ld", &*qmax, &*qbase, &*qincr);
  fclose(db_info);
  }
void create_tables()
  {
/* create query modification tables */
  exec sql create table posqm
         (e_num integer2, snum integer2, level integer1, keyno integer2,
          accinfo c86);
  exec sql create table empqm
         (e_num integer2, dnum integer2, ename c20, address c70,
          salary integer4, title c30, jobdesc c60);
  exec sql create table skillqm
         (snum integer2, sname c20, stype c34);
/* create semi-materialization tables */
  exec sql create table possm
         (e_num integer2, snum integer2, level integer1, keyno integer2,
          accinfo c86);
  exec sql create table empsm
         (e_num integer2, dnum integer2, ename c20, address c70,
          salary integer4, title c30, jobdesc c60);
  exec sql create table skillsm
         (snum integer2, sname c20, stype c34);
```

```
  exec sql create table pos_prim
          (e_num integer2, keyno integer2);
  exec sql create table emp_prim
          (e_num integer2, ename c20, salary integer4);
/* create full materialization tables */
  exec sql create table posfm
          (e_num integer2, snum integer2, level integer1, keyno integer2,
           accinfo c86);
  exec sql create table empfm
          (e_num integer2, dnum integer2, ename c20, address c70,
           salary integer4, title c30, jobdesc c60);
  exec sql create table skillfm
          (snum integer2, sname c20, stype c34);
  exec sql create table full_mat
          (e_num integer2, ename c20, salary integer4, keyno integer2);
  }
void create_views(int viewcut)
{
    exec sql begin declare section;
       int view_cut;
    exec sql end declare section;
    view_cut = viewcut;
    exec sql create view full_view(e_num, ename, salary, keyno) as
           select empqm.e_num, empqm.ename, empqm.salary, outer2.keyno
           from empqm, posqm outer2
           where exists
           (select *
            from posqm inner2
            where empqm.e_num = inner2.e_num
            and outer2.keyno = inner2.keyno
            and inner2.level >= :view_cut);
```

```
    exec sql create view sm_view(e_num, ename, salary, keyno) as
            select emp_prim.e_num, emp_prim.ename, emp_prim.salary, outer4.keyno
            from emp_prim, pos_prim outer4
            where exists
            (select *
             from pos_prim inner4
             where emp_prim.e_num = inner4.e_num
             and outer4.keyno = inner4.keyno);
}
void create_update_table()
  {
  exec sql create table update_tbl
          (e_num integer2, snum integer2, level integer1, keyno integer2,
           accinfo c86);
  exec sql copy table update_tbl
          (e_num = c0colon, snum= c0colon, level = c0colon,
           keyno = c0colon, accinfo = c0nl)
           from :updatinfo;
  }
void copy_base_tables()
  {
  exec sql copy table posqm
          (e_num = c0colon, snum = c0colon, level = c0colon, keyno = c0colon,
           accinfo = c0nl)
           from :posinfo;
  exec sql copy table possm
          (e_num = c0colon, snum = c0colon, level = c0colon, keyno = c0colon,
           accinfo = c0nl)
           from :posinfo;
  exec sql copy table posfm
          (e_num = c0colon, snum = c0colon, level = c0colon, keyno = c0colon,
           accinfo = c0nl)
           from :posinfo;
  exec sql copy table empqm
          (e_num = c0colon, dnum = c0colon, ename = c0colon, address = c0colon,
           salary = c0colon, title = c0colon, jobdesc = c0nl)
           from :empinfo;
```

```
exec sql copy table empsm
        (e_num = c0colon, dnum = c0colon, ename = c0colon, address = c0colon,
         salary = c0colon, title = c0colon, jobdesc = c0nl)
         from :empinfo;
exec sql copy table empfm
        (e_num = c0colon, dnum = c0colon, ename = c0colon, address = c0colon,
         salary = c0colon, title = c0colon, jobdesc = c0nl)
         from :empinfo;
exec sql copy table skillqm
        (snum = c0colon, sname = c0colon, stype = c0nl)
         from :skilinfo;
exec sql copy table skillsm
        (snum = c0colon, sname = c0colon, stype = c0nl)
         from :skilinfo;
exec sql copy table skillfm
        (snum = c0colon, sname = c0colon, stype = c0nl)
         from :skilinfo;
}
void copy_semi_n_full_mats(int viewcut)
{
exec sql begin declare section;
int view_cut;
exec sql end declare section;
view_cut = viewcut;
exec sql insert into pos_prim (e_num, keyno)
          select e_num, keyno
          from possm
          where level >= :view_cut;
exec sql insert into emp_prim (e_num, ename, salary)
          select e_num, ename, salary
          from empsm;
```

```
exec sql insert into full_mat (e_num, ename, salary, keyno)
      select empfm.e_num, empfm.ename, empfm.salary, outer2.keyno
      from empfm, posfm outer2
      where exists
      (select *
       from posfm inner2
       where empfm.e_num = inner2.e_num
       and outer2.keyno = inner2.keyno
       and inner2.level >= :view_cut);
 }
void create_table_index()
 {
 exec sql modify empqm to cbtree on e_num;
 exec sql modify empsm to cbtree on e_num;
 exec sql modify empfm to cbtree on e_num;
 exec sql modify posqm to cbtree on level;
 exec sql modify possm to cbtree on level;
 exec sql modify posfm to cbtree on level;
 exec sql modify emp_prim to cbtree on salary;
 exec sql modify pos_prim to cbtree on e_num;
 exec sql modify full_mat to cbtree on salary;
     /*  create secondary indexes  */
 exec sql create index empqmdx
        on empqm (e_num);
 exec sql create index empsmdx
        on empsm (e_num);
 exec sql create index empfmdx
        on empfm (e_num);
 exec sql create index posqmdx
        on posqm (level);
 exec sql create index possmdx
        on possm (level);
 exec sql create index posfmdx
        on posfm (level);
 exec sql create index e_primdx
        on emp_prim (salary);
 exec sql create index p_primdx
        on pos_prim(e_num);
```

```
  exec sql create index f_matdx
         on full_mat (salary);
  )
void module_qm(char cntrl_char, int viewcut, long querycut, double *timeqm,
            FILE *run_rslt)
  {
  clock_t tstart = 0, tstop = 0;
  double elap_time;
  long tbl_cnt = 0;
  exec sql begin declare section;
    int view_cut;
    long query_cut;
    long qnum;
    char qname[21];
    long qkeyno;
  exec sql end declare section;
  exec sql declare qm_cl cursor for
  select e_num, ename, keyno
    from full_view
    where salary >= :query_cut;
  view_cut = viewcut;
  query_cut = querycut;
  switch(cntrl_char)
    {
    case 'K':
      create_update_table();
      exec sql insert into posqm
              select *
               from update_tbl;
      exec sql drop update_tbl;
      break;
    case 'Q':
      tstart = clock();
      exec sql open qm_cl;
      exec sql whenever not found goto closeqm_cl;
```

```
    while(sqlca.sqlcode == 0)
     {
     exec sql fetch qm_cl
            into :qnum, :qname, :qkeyno;
    /*  printf("\nnumber = %d", qnum); */
     tbl_cnt++;
     }
    closeqm_cl:
     exec sql whenever not found continue;
     tstop = clock();
     exec sql close qm_cl;
    break;
  default:
    printf("\nIncorrect control character\n");
    break;
  }
 elap_time = (tstop - tstart)/(double)CLK_TCK;
 *timeqm = *timeqm + elap_time;
 write_run_result('q', cntrl_char, viewcut, querycut, elap_time, tbl_cnt,
              run_rslt);
 }
void module_sm(char cntrl_char, int viewcut, long querycut, double *timesm,
             FILE *run_rslt)
 {
 clock_t tstart = 0, tstop = 0;
 double elap_time;
 long tbl_cnt = 0;
 exec sql begin declare section;
   int view_cut;
   long query_cut;
   long snum;
   char sname[21];
   long skeyno;
 exec sql end declare section;
```

```
exec sql declare sm_cl cursor for
select e_num, ename, keyno
  from sm_view
  where salary >= :query_cut;
view_cut = viewcut;
query_cut = querycut;
switch(cntrl_char)
  {
  case 'K':
    create_update_table();
    exec sql insert into possm
            select *
            from update_tbl;
    tstart = clock();
    exec sql insert into pos_prim
            select e_num, keyno
             from update_tbl
             where level >= :view_cut;
    tstop = clock();
    exec sql drop update_tbl;
    break;
  case 'Q':
    tstart = clock();
    exec sql open sm_cl;
    exec sql whenever not found goto closesm_cl;
    while (sqlca.sqlcode == 0)
     {
     exec sql fetch sm_cl
             into :snum, :sname, :skeyno;
    /*  printf("\nsnum = %d", snum); */
     tbl_cnt++;
     }
    closesm_cl:
     exec sql whenever not found continue;
     tstop = clock();
     exec sql close sm_cl;
    break;
```

```
    default:
      printf("\Nincorrect control character\n");
      break;
    }
  elap_time = (tstop - tstart)/(double)CLK_TCK;
  *timesm = *timesm + elap_time;
  write_run_result('s', cntrl_char, viewcut, querycut, elap_time, tbl_cnt,
              run_rslt);
  }
void module_fm(char cntrl_char, int viewcut, long querycut, double *timefm,
             FILE *run_rslt)
  {
  clock_t tstart = 0, tstop = 0;
  double elap_time;
  long qcnt = 0;
  exec sql begin declare section;
    int view_cut;
    long query_cut;
    long tbl_cnt;
    long fnum;
    char fname[21];
    long fkeyno;
  exec sql end declare section;
  exec sql declare fm_cl cursor for
  select e_num, ename, keyno
    from full_mat
    where salary >= :query_cut;
  view_cut = viewcut;
  query_cut = querycut;
  switch(cntrl_char)
    {
```

```
case 'K':
  create_update_table();
  exec sql insert into posfm
          select *
          from update_tbl;
  tstart = clock();
  exec sql drop full_mat;
  exec sql create table full_mat
      (e_num integer2, ename c20, salary integer4, keyno integer2);
  exec sql insert into full_mat (e_num, ename, salary, keyno)
        select empfm.e_num, empfm.ename, empfm.salary, outer.keyno
        from empfm, posfm outer
        where exists
          (select *
           from posfm inner
            where empfm.e_num = inner.e_num
            and outer.keyno = inner.keyno
            and inner.level >= :view_cut);
  exec sql modify full_mat to cbtree on salary;
  tstop = clock();
  exec sql drop update_tbl;
  break;
case 'Q':
  tstart = clock();
  exec sql open fm_cl;
  exec sql whenever not found goto closefm_cl;
  while (sqlca.sqlcode == 0)
   {
   exec sql fetch fm_cl
           into :fnum, :fname, :fkeyno;
   /*  printf("\n fnum = %d", fnum); */
   qcnt++;
   }
  closefm_cl:
   exec sql whenever not found continue;
   tstop = clock();
   exec sql close fm_cl;
  break;
```

```
    default:
      printf("\Nincorrect control character\n");
      break;
    }
  elap_time = (tstop - tstart)/(double)CLK_TCK;
  *timefm = *timefm + elap_time;
  exec sql select rowtot = count(e_num)
          into :tbl_cnt
          from full_mat
          where salary >= :query_cut;
  write_run_result('f', cntrl_char, viewcut, querycut, elap_time, tbl_cnt,
                run_rslt);
  }
void write_file_headings(char* param, char* updt_tbl, FILE* fresult_fl,
                     FILE* run_rslt)
  {
  time_t today_t;
  time(&today_t);
  fprintf(fresult_fl,"\n  %s  -  FINAL RESULTS (vsgxpdp7) -\n", ctime(&today_t));
  fprintf(fresult_fl,"\n The %s is the parameter being tested", param);
  fprintf(fresult_fl,"\n The %s table is the table being updated", updt_tbl);
  fprintf(run_rslt,"\n  %s  -  RUN RESULTS (vsgxpdp7) -\n", ctime(&today_t));
  fprintf(run_rslt,"\n The %s is the parameter being tested", param);
  fprintf(run_rslt,"\n The %s table is the table being updated\n", updt_tbl);
  }
void write_run_result(char strat, char cntrl_char, int viewcut, long querycut,
                   double elap_time, long tbl_cnt, FILE *run_rslt)
  {
  printf("\n%cm cc=%c vc=%d qc=%ld et=%.2lf tc=%ld", strat, cntrl_char,
       viewcut, querycut, elap_time, tbl_cnt);
  fprintf(run_rslt,"\n%cm cc=%c vc=%d qc=%ld et=%.2lf tc=%ld",strat,
        cntrl_char, viewcut, querycut, elap_time, tbl_cnt);
  }
```

```
void write_final_result(int run, int viewcut, long querycut, int updt_siz,
                    long countb, long countv, long countq, float fv,
                    float fva, float fq, float fqa, float P,
                    double timeqm, double timesm, double timefm,
                    FILE *fresult_fl, FILE *run_rslt)
  {
  printf("\n\nRUN# %d, VCUT= %d, QCUT= %ld, #TUP= %d, BASE= %ld, VIEW= %ld,"\
        " QUERY= %ld", run, viewcut, querycut, updt_siz, countb, countv,
        countq);
  printf("\nFV= %.2f, FVA= %f, FQ= %.2f, FQA= %f P= %.2f",fv, fva, fq,
        fqa, P);
  printf("\nTIMEQM= %.3lf sec, TIMESM= %.3lf sec, TIMEFM= %.3lf sec\n",
       timeqm, timesm, timefm);
  fprintf(fresult_fl,"\n\NRUN# %d, VCUT= %d, QCUT= %ld, #TUP= %d, BASE= %ld,"\
       " VIEW= %ld, QUERY= %ld", run, viewcut, querycut, updt_siz, countb,
         countv, countq);
  fprintf(fresult_fl,"\NFV= %.2f, FVA= %f, FQ= %.2f, FQA= %f P= %.2f",fv, fva,
        fq, fqa, P);
  fprintf(fresult_fl,"\NTIMEQM= %.3lf sec, TIMESM= %.3lf sec, TIMEFM= %.3lf"\
         " sec\n",timeqm, timesm, timefm);
  fprintf(run_rslt,"\n\NRUN# %d, VCUT= %d, QCUT= %ld, #TUP= %d, BASE= %ld,"\
        " VIEW= %ld, QUERY= %ld", run, viewcut, querycut, updt_siz, countb,
          countv, countq);
  fprintf(run_rslt,"\NFV= %.2f, FVA= %f, FQ= %.2f, FQA= %f P= %.2f", fv, fva,
         fq, fqa, P);
  fprintf(run_rslt,"\NTIMEQM= %.3lf sec, TIMESM= %.3lf sec, TIMEFM= %.3lf"\
         " sec\n",timeqm, timesm, timefm);
  }
void compute_avg_time(int Q, double *timeqm, double *timesm, double *timefm)
  {
  if(Q > 0)
    {
    *timeqm = *timeqm / (double)Q;
    *timesm = *timesm / (double)Q;
    *timefm = *timefm / (double)Q;
    }
```

```
  else
    {
    printf("\n\NERROR: dividing times by 0,  **** results are VOID ****\n");
    }
  }
void compute_fv_and_fq_and_P(int vmax, int vbase, int vincr, int vcut,
                              float *fv,long qmax, long qbase, long qincr,
                              long qcut, float *fq, int K, int Q, float *P)
  {
  *fv = (float)(vmax) - ((float)(vcut - vbase) / (float)(vincr));
  *fv = (*fv + (float)(vincr) / (float)(vincr)) / (float)(vmax);
  *fq = (float)(qmax) - ((float)(qcut - qbase) / (float)(qincr));
  *fq = (*fq + (float)(qincr) / (float)(qincr)) / (float)(qmax);
  *P  = (float)(K)/(float)(K + Q);
  }
void compute_table_counts(long *countb, long *countv, long *countq,
                           long querycut, float *fva, float *fqa)
  {
  exec sql begin declare section;
    long query_cut;
    long tbl_cnt;
  exec sql end declare section;
  query_cut = querycut;
  exec sql create table base_mat
           (e_num integer2, ename c20, salary integer4, keyno integer2);
  exec sql insert into base_mat (e_num, ename, salary, keyno)
             select empfm.e_num, empfm.ename, empfm.salary, posfm.keyno
             from empfm, posfm
             where empfm.e_num = posfm.e_num;
  exec sql select rowtot = count(e_num)
           into :tbl_cnt
           from base_mat;
  *countb = tbl_cnt;
  exec sql select rowtot = count(e_num)
           into :tbl_cnt
           from full_mat;
  *countv = tbl_cnt;
```

```
exec sql select rowtot = count(e_num)
         into :tbl_cnt
         from full_mat
         where salary >= :query_cut;
*countq = tbl_cnt;
*fva = (float)((double)*countv / (double)*countb);
*fqa = (float)((double)*countq / (double)*countv);
exec sql drop base_mat;
}
void refresh_update_text_file(long card, long i, long update_siz)
{
long update_base;
int num_of_fields, j, change_field = 4;
char file_name[size] = updatinfo, *file_ptr;
FILE *updat_fl;
     struct field_attrib
  {
    char  field_type;
    int   field_width;
    char  field_info;
    long  lower_bound;
    int   increment;
    long  upper_bound;
    struct       field_attrib  *next;
   };
struct field_attrib *first_field = NULL;
struct field_attrib *current_field = NULL;
struct field_attrib *print_ptr = NULL;
file_ptr = &file_name[0];
update_base = (i * update_siz) + card +1;  /* compute new key base number */
```

```
   /**  Read old control input for data generation program  **/
updat_fl = fopen(update_file, "r");
if(!updat_fl)
  {
  printf("\NERROR: update control file did not open to read");
  fcloseall();
  exec sql disconnect;
  exit(1);
  }
fscanf(updat_fl, "%*d\n");
fscanf(updat_fl, "%d\n", &num_of_fields);
fscanf(updat_fl, "%*s\n");
for (j = 1; j <= num_of_fields; j++)
  {
  if (j == 1)
    {
    first_field = (struct field_attrib*)malloc(sizeof(struct field_attrib));
    if (first_field == NULL) printf("\NERROR: Memory did not allocate!!!");
    current_field = first_field;
    }
  else
    {
    current_field->next =(struct field_attrib*)malloc(sizeof(struct field_attrib));
    current_field = current_field->next;
    }
  current_field->next = NULL;
  fscanf(updat_fl, "\n%c\n", &current_field->field_type);
  fscanf(updat_fl, "%d\n", &current_field->field_width);
  fscanf(updat_fl, "%c\n", &current_field->field_info);
  fscanf(updat_fl, "%ld\n", &current_field->lower_bound);
  fscanf(updat_fl, "%d\n", &current_field->increment);
  fscanf(updat_fl, "%ld\n", &current_field->upper_bound);
  if (j == change_field)      /* changing base for keyno field */
     {
     current_field->lower_bound = update_base;
     }
}
fclose(updat_fl);
```

```
    /**  write updated control input for data generation program  **/
updat_fl = fopen(update_file, "w");
if(!updat_fl)
  {
  printf("\NERROR: update control file did not open to write");
  fcloseall();
  exec sql disconnect;
  exit(1);
  }
fprintf(updat_fl, "%ld\n", update_siz);
fprintf(updat_fl, "%d\n", num_of_fields);
fprintf(updat_fl, "%s", file_ptr);
print_ptr = first_field;
while(print_ptr != NULL)
  {
  fprintf(updat_fl, "\n\n%c\n", print_ptr->field_type);
  fprintf(updat_fl, "%d\n", print_ptr->field_width);
  fprintf(updat_fl, "%c\n", print_ptr->field_info);
  fprintf(updat_fl, "%ld\n", print_ptr->lower_bound);
  fprintf(updat_fl, "%d\n", print_ptr->increment);
  fprintf(updat_fl, "%ld", print_ptr->upper_bound);
  print_ptr = print_ptr->next;
  }
fclose(updat_fl);
system("datagen");
}
```

LIST OF REFERENCES

1. Date, C. J., *An Introduction to Database Systems*, 3rd Ed., Addison-Wesley Publishing Company, 1981.

2. Kamel,M., Davidson, S., *Semi-materialization: a technnique for optimizing frequently executed queries*, Data and Knowledge Engineering 6, North-Holland, 1991.

3. South, J., *A performance Analysis of View Materializations Strategies for Select-Project-Join Expressions*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1991.

4. Kamel, M., Davidson, S., *Semi-materialization: A Performance Analysis*, University of Pennsylvania, 1987.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Library, Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
2.	Administrative Sciences Department Naval Postgraduate School Attn: Prof. M. N. Kamel, Code AS/KA Monterey, CA 93943-5000	1
3.	University of Rochester U.S. Navy ROTC Unit Attn: CDR R. Griffin Rochester, NY 14627-0016	1
4.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
5.	LT Curtis G. Barefield Jr. Department Head Class 121 Surface Warfare Officer School Command Newport, RI 02841-5012	3

Thesis
B21757 Barefield
c.1 A performance analysis
of view materialization
strategies for general
expressions.